

-

整理自互联网，仅供学习交流。



2018-10-30

[公司名称]

[公司地址]

目录

[一、 BAT薪资、技能要求、开发流程 1](#_Toc6602334)

[1.1 岗位、薪资、建议 1](#_Toc6602335)

[1.1.1 百度 1](#_Toc6602336)

[1.1.2 阿里 1](#_Toc6602337)

[1.1.3 腾讯 2](#_Toc6602338)

[1.1.4 腾讯的薪酬和级别的关系 2](#_Toc6602339)

[1.1.5 BAT招聘渠道 3](#_Toc6602340)

[1.1.6 我的建议 4](#_Toc6602341)

[1.2 阿里P系列技术要求(P5-P7) 4](#_Toc6602342)

[1.2.1 阿里P5(高级研发工程师) 5](#_Toc6602343)

[1.2.2 阿里P6(技术主管) 6](#_Toc6602344)

[1.2.3 阿里P7(技术专家) 6](#_Toc6602345)

[1.3 产品技术人必备干货：产品开发流程（完整版） 7](#_Toc6602346)

[1.3.1 产品概念设计：MRD 7](#_Toc6602347)

[1.3.2 二：原型设计(交互设计) 8](#_Toc6602348)

[1.3.3 三： 视觉设计 11](#_Toc6602349)

[1.3.4 四：PRD详细需求文档 11](#_Toc6602350)

[1.3.5 五：研发阶段 12](#_Toc6602351)

[二、 BAT求职 13](#_Toc6602352)

[2.1 5步教你成功求职进入BAT 14](#_Toc6602353)

[2.1.1 如何写简历 14](#_Toc6602354)

[2.1.2 如何选择求职渠道？ 15](#_Toc6602355)

[2.1.3 如何现场面试? 18](#_Toc6602356)

[2.1.4 如何选择Offer? 18](#_Toc6602357)

[2.1.5 如何选择公司 19](#_Toc6602358)

[2.2 手把手教你求职进入BAT 19](#_Toc6602359)

[2.2.1 求职跳槽的三大误区 19](#_Toc6602360)

[2.2.2 第一步：简历自我评价改进 21](#_Toc6602361)

[2.2.3 第二步：简历项目经验改进 22](#_Toc6602362)

[2.2.4 第三步：定制公司的面试简历 24](#_Toc6602363)

[2.3 如何提高面试成功率 25](#_Toc6602364)

[2.3.1 第一步 快速聚焦岗位职责 25](#_Toc6602365)

[2.3.2 第二步 准备你的亮点 25](#_Toc6602366)

[2.3.3 第三步 展示你强大的自信 25](#_Toc6602367)

[2.4 前BAT面试官详谈进入BAT面试经验 26](#_Toc6602368)

[2.4.1 1.从自我介绍开始，全程保持微笑 26](#_Toc6602369)

[2.4.2 2.技术面试 27](#_Toc6602370)

[2.4.3 3.项目介绍 27](#_Toc6602371)

[2.4.4 4.职业规划 28](#_Toc6602372)

[2.5 怎样跳槽最有利于自身的发展？ 28](#_Toc6602373)

[2.5.1 主动求发展 28](#_Toc6602374)

[2.5.2 发展遇到瓶颈 28](#_Toc6602375)

[2.5.3 遇见不靠谱老板 29](#_Toc6602376)

[三、 阿里P8架构师谈： 30](#_Toc6602377)

[3.1 HashMap的底层结构、原理、扩容机制 30](#_Toc6602378)

[3.1.1 摘要 30](#_Toc6602379)

[3.1.2 简介 30](#_Toc6602380)

[3.1.3 线程安全性 40](#_Toc6602381)

[3.1.4 JDK1.8与JDK1.7的性能对比 43](#_Toc6602382)

[四、 高并发编程系列 47](#_Toc6602383)

[4.1 ConcurrentHashMap的实现原理(JDK1.7和JDK1.8) 47](#_Toc6602384)

[4.1.1 哈希表 48](#_Toc6602385)

[4.1.2 ConcurrentHashMap与HashMap等的区别 49](#_Toc6602386)

[4.1.3 JDK1.7版本的CurrentHashMap的实现原理 50](#_Toc6602387)

[4.1.4 JDK1.8版本的CurrentHashMap的实现原理 52](#_Toc6602388)

[4.1.5 总结 53](#_Toc6602389)

[4.2 NIO、BIO、AIO的区别，及NIO的应用和框架选型 53](#_Toc6602390)

[4.2.1 AIO、BIO、NIO的区别 53](#_Toc6602391)

[4.2.2 NIO的3个核心概念 55](#_Toc6602392)

[4.2.3 NIO的应用和框架 56](#_Toc6602393)

[4.3 4大并发工具类详解CountDownLatch、Semaphore等 58](#_Toc6602394)

[4.3.1 并发工具包 58](#_Toc6602395)

[4.3.2 常用的并发容器 59](#_Toc6602396)

[4.3.3 常用的并发工具类 61](#_Toc6602397)

[4.4 4种常用Java线程锁的特点，性能比较、使用场景 65](#_Toc6602398)

[4.4.1 多线程的缘由 65](#_Toc6602399)

[4.4.2 多线程并发面临的问题 65](#_Toc6602400)

[4.4.3 4种Java线程锁(线程同步) 66](#_Toc6602401)

[4.4.4 Java线程锁总结 69](#_Toc6602402)

[4.5 Java锁详解：独享锁/共享锁+公平锁/非公平锁+乐观锁/悲观锁 70](#_Toc6602403)

[4.5.1 乐观锁 VS 悲观锁 71](#_Toc6602404)

[4.5.2 公平锁 VS 非公平锁 73](#_Toc6602405)

[4.5.3 独享锁 VS 共享锁 74](#_Toc6602406)

[4.5.4 分段锁 75](#_Toc6602407)

[4.6 4大JVM性能分析工具详解，及内存泄漏分析方案 75](#_Toc6602408)

[4.6.1 谈到性能优化分析一般会涉及到： 75](#_Toc6602409)

[4.6.2 什么场景需要JVM调优 76](#_Toc6602410)

[4.6.3 JVM性能监控分析工具 76](#_Toc6602411)

[4.6.4 VisualVM 77](#_Toc6602412)

[4.6.5 Jconsole 80](#_Toc6602413)

[4.6.6 MAT 86](#_Toc6602414)

[4.6.7 JVM内存泄漏分析 91](#_Toc6602415)

[4.7 并发容器的原理，7大并发容器详解、及使用场景 92](#_Toc6602416)

[4.7.1 并发容器的由来 92](#_Toc6602417)

[4.7.2 同步容器面临的问题 94](#_Toc6602418)

[4.7.3 什么是并发容器 94](#_Toc6602419)

[4.7.4 并发容器有哪些分类 95](#_Toc6602420)

[4.7.5 ConcurrentHashMap的实现 98](#_Toc6602421)

[4.8 Java线程池的使用方式，核心运行原理、以及注意事项 99](#_Toc6602422)

[4.8.1 线程池的缘由 99](#_Toc6602423)

[4.8.2 线程池的处理流程 100](#_Toc6602424)

[4.8.3 线程池的使用(ThreadPoolExecutor) 101](#_Toc6602425)

[4.8.4 线程池的注意事项 105](#_Toc6602426)

[五、 高并发架构系列 107](#_Toc6602427)

[5.1 Redis并发竞争key的解决方案详解 108](#_Toc6602428)

[5.1.1 Redis高并发的问题 108](#_Toc6602429)

[5.1.2 如何解决Redis的并发竞争key问题 108](#_Toc6602430)

[**5.2** **Redis缓存和MySQL数据一致性方案详解** 110](#_Toc6602431)

[5.2.1 需求起因 110](#_Toc6602432)

[5.2.2 第一种方案：采用延时双删策略 112](#_Toc6602433)

[5.2.3 第二种方案：异步更新缓存(基于订阅binlog的同步机制) 113](#_Toc6602434)

[5.3 解决Redis缓存雪崩、缓存穿透、缓存并发等5大难题 114](#_Toc6602435)

[5.3.1 缓存解决方案： 114](#_Toc6602436)

[5.3.2 缓存雪崩 114](#_Toc6602437)

[5.3.3 缓存穿透 116](#_Toc6602438)

[5.3.4 缓存并发 117](#_Toc6602439)

[5.3.5 缓存预热 117](#_Toc6602440)

[5.4 **大型网站Web前端优化最佳实践，以及最全优化工具集锦** 117](#_Toc6602441)

[5.4.1 大型网站Web优化最佳实践 117](#_Toc6602442)

[5.4.2 YSlow Web优化最佳实践 124](#_Toc6602443)

[5.4.3 Web性能优化工具 127](#_Toc6602444)

[5.5 负载均衡的原理、分类、实现架构，以及使用场景 128](#_Toc6602445)

[5.5.1 为什么需要负载均衡 129](#_Toc6602446)

[5.5.2 负载均衡的原理 129](#_Toc6602447)

[5.5.3 负载均衡的作用 130](#_Toc6602448)

[5.5.4 负载均衡的分类 130](#_Toc6602449)

[5.5.5 最常见的四层和七层负载均衡 131](#_Toc6602450)

[5.5.6 负载均衡应用场景 132](#_Toc6602451)

[5.6 常见分布式文件存储介绍、选型比较、架构设计 133](#_Toc6602452)

[5.6.1 数据正成为世界上最有价值的资源，分布式文件存储是应对数据爆炸的最好解决方案，那就会涉及到分布式文件存储方案、选型、架构设计等。 134](#_Toc6602453)

[5.6.2 分布式文件存储的来源 134](#_Toc6602454)

[5.6.3 常见的分布式文件系统 134](#_Toc6602455)

[5.6.4 知名开源分布式文件存储 135](#_Toc6602456)

[5.6.5 典型的分布式文件存储的架构设计 136](#_Toc6602457)

[5.6.6 分布式存储的未来 138](#_Toc6602458)

[5.7 开源搜索引擎Lucene、Solr、Sphinx等优劣势比较 139](#_Toc6602459)

[5.7.1 开源搜索引擎分类 139](#_Toc6602460)

[5.7.2 Lucene 139](#_Toc6602461)

[5.7.3 Apache Solr 140](#_Toc6602462)

[5.7.4 2.Solr的优缺点 141](#_Toc6602463)

[5.7.5 Elastic Search 141](#_Toc6602464)

[5.7.6 Elasticsearch 与 Solr 的比较总结 142](#_Toc6602465)

[5.7.7 Sphinx 143](#_Toc6602466)

[5.8 如何从0到1设计一个类Dubbo的RPC框架 144](#_Toc6602467)

[5.8.1 之前分享了如何从0到1设计一个MQ消息队列，今天谈谈“如何从0到1设计一个Dubbo的RPC框架”，重点考验： 144](#_Toc6602468)

[5.8.2 RPC和RPC框架 144](#_Toc6602469)

[5.8.3 Dubbo的发展由来 144](#_Toc6602470)

[5.8.4 RPC框架的核心设计 146](#_Toc6602471)

[5.8.5 RPC框架涉及技术 149](#_Toc6602472)

[六、 职场发展 151](#_Toc6602473)

[6.1 互联网寒冬，程序员如何突破重围？我的3个建议 151](#_Toc6602474)

[6.1.1 23-27岁：技术积累阶段 152](#_Toc6602475)

[6.1.2 27-30岁：形成思维方法论和知识体系的阶段 152](#_Toc6602476)

[6.1.3 30-33岁这个阶段，最关键的是了解你自己。 153](#_Toc6602477)

[6.2 **从职业方向，谈程序员如何突破成长瓶颈？** 153](#_Toc6602478)

[6.3 技术人如何提升自己的核心竞争力 156](#_Toc6602479)

[6.4 **又一名华为中年程序员倒下了，如何打破35岁程序员的魔咒？** 159](#_Toc6602480)

[6.5 **技术人成长的三大原则** 161](#_Toc6602481)

[6.6 **技术人的成功之路** 165](#_Toc6602482)

[七、 技术人的未来 167](#_Toc6602483)

[7.1 首先，认清外在误区 167](#_Toc6602484)

[7.2 其次，遵循发展周期 168](#_Toc6602485)

[7.3 认清自己的优势 169](#_Toc6602486)

[7.4 站在巨人的肩膀上，继续前行 170](#_Toc6602487)

[7.5 程序员真的只能干到35岁？——我的35岁危机度过之道！ 170](#_Toc6602488)

[7.5.1 危机的根源 171](#_Toc6602489)

[7.5.2 做正确的事情 171](#_Toc6602490)

[7.5.3 23-27岁：技术积累阶段 171](#_Toc6602491)

[7.5.4 27-30岁：形成思维方法论和知识体系的阶段 172](#_Toc6602492)

[7.5.5 走自己的路 172](#_Toc6602493)

[7.6 程序员怎么升职加薪？ 173](#_Toc6602494)

[7.6.1 程序员升职加薪之路 173](#_Toc6602495)

[7.6.2 技术线路升职之路 174](#_Toc6602496)

[7.6.3 产品运营转型之路 175](#_Toc6602497)

[7.7 如何选择一家公司 175](#_Toc6602498)

[7.7.1 BAT 175](#_Toc6602499)

[7.7.2 成长型公司 177](#_Toc6602500)

[7.8 Java程序员的发展之路和职业规划 178](#_Toc6602501)

[7.8.1 技术人的困扰 178](#_Toc6602502)

[7.8.2 认清外在误区 179](#_Toc6602503)

[7.8.3 遵循发展周期 179](#_Toc6602504)

[7.8.4 找到自己的方向 180](#_Toc6602505)

[7.8.5 站在巨人的肩膀上 180](#_Toc6602506)

[八、 数据库 181](#_Toc6602507)

[8.1 数据库主从同步的3种一致性方案实现，及优劣比较 181](#_Toc6602508)

[8.1.1 数据主从同步的由来 181](#_Toc6602509)

[8.1.2 数据同步一致性解决方案 182](#_Toc6602510)

[8.1.3 缓存记录写key法 185](#_Toc6602511)

[8.2 NoSQL和SQL的区别，NoSQL的使用场景和选型比较 186](#_Toc6602512)

[8.2.1 什么是NoSQL 186](#_Toc6602513)

[8.2.2 SQL 和 NoSQL 的区别 186](#_Toc6602514)

[8.2.3 SQL和Nosql的选型和比较 187](#_Toc6602515)

[8.2.4 NoSQL的常见类型和比较 188](#_Toc6602516)

[8.3 MySQL慢查询优化、索引优化、以及表等优化总结 190](#_Toc6602517)

[8.3.1 MySQL优化概述 190](#_Toc6602518)

[8.3.2 MySQL优化方案 190](#_Toc6602519)

[8.3.3 索引优化 191](#_Toc6602520)

[8.4 MySQL有哪些存储引擎，各自的优缺点，应用场景 195](#_Toc6602521)

[8.4.1 MySQL常用存储引擎介绍 195](#_Toc6602522)

[8.4.2 MySQL存储引擎MyISAM与InnoDB如何选择 196](#_Toc6602523)

[8.5 分布式锁的3种实现（数据库、缓存、Zookeeper) 197](#_Toc6602524)

[8.5.1 分布式锁的几种实现方式 197](#_Toc6602525)

[8.5.2 基于数据库实现分布式锁 198](#_Toc6602526)

[8.5.3 基于缓存实现分布式锁 202](#_Toc6602527)

[8.5.4 基于Zookeeper实现分布式锁 203](#_Toc6602528)

[8.5.5 三种方案的比较 205](#_Toc6602529)

[8.6 MySQL数据库的索引原理、与慢SQL优化的5大原则 206](#_Toc6602530)

[8.6.1 MySQL索引原理 206](#_Toc6602531)

[8.6.2 2.索引原理 206](#_Toc6602532)

[8.6.3 慢查询优化基本步骤 210](#_Toc6602533)

[8.7 分布式事务的解决方案，以及原理、总结 219](#_Toc6602534)

[8.7.1 事务 219](#_Toc6602535)

[8.7.2 分布式事务 220](#_Toc6602536)

[8.7.3 常见的分布式事务解决方案 222](#_Toc6602537)

[8.7.4 分布式事务总结 228](#_Toc6602538)

[8.8 MySQL行锁、表锁、悲观锁、乐观锁的特点与应用 228](#_Toc6602539)

[8.8.1 MySQL锁概述 229](#_Toc6602540)

[8.8.2 行锁 和 表锁 229](#_Toc6602541)

[8.8.3 悲观锁 和 乐观锁 230](#_Toc6602542)

[8.8.4 共享锁 230](#_Toc6602543)

[8.8.5 表锁和行锁应用场景： 231](#_Toc6602544)

[8.9 架构设计之数据库垂直、水平拆分六大原则 231](#_Toc6602545)

[8.9.1 数据拆分前其实是要首先做准备工作的，然后才是开始数据拆分，我先讲拆分前需要做的事情： 231](#_Toc6602546)

[8.9.2 第一步，首选垂直拆分 232](#_Toc6602547)

[8.9.3 第二步：其次才是水平拆分 232](#_Toc6602548)

[8.9.4 总结，数据库拆分原则： 233](#_Toc6602549)

[8.10 MongoDB、Hbase、Redis等NoSQL优劣势、应用场景 233](#_Toc6602550)

[8.10.1 NoSQL的四大种类 233](#_Toc6602551)

[8.10.2 主流nosql的详解：MongoDB、Hbase、Redis 234](#_Toc6602552)

[8.10.3 MongoDB 235](#_Toc6602553)

[8.10.4 HBase 236](#_Toc6602554)

[8.10.5 Redis 238](#_Toc6602555)

[8.11 数据库分库分表、读写分离的原理实现，使用场景 239](#_Toc6602556)

[8.11.1 为什么要分库分表和读写分离？ 239](#_Toc6602557)

[8.11.2 分库分表的原理和实现 240](#_Toc6602558)

[8.11.3 读写分离的原理和实现 242](#_Toc6602559)

[8.11.4 分库分表、读写分离总结： 243](#_Toc6602560)

[8.12 如何实现分库分表+动态数据源+读写分离 244](#_Toc6602561)

[8.12.1 分布式数据层中间件： 244](#_Toc6602562)

[8.12.2 常见的数据层中间件： 245](#_Toc6602563)

[8.12.3 分布式数据层中间件：具体实现 247](#_Toc6602564)

[8.12.4 连接池及SQL监控 259](#_Toc6602565)

[九、 BAT必考JVM系列专题 260](#_Toc6602566)

[9.1 JVM性能调优的6大步骤，及关键调优参数详解 260](#_Toc6602567)

[9.1.1 JVM内存调优 261](#_Toc6602568)

[9.1.2 JVM性能调优方法和步骤 261](#_Toc6602569)

[9.2 深入详解JVM内存模型与JVM参数详细配置 265](#_Toc6602570)

[9.2.1 JVM内存结构 266](#_Toc6602571)

[9.2.2 堆内存（Heap） 266](#_Toc6602572)

[9.3 JVM的4种垃圾回收算法、垃圾回收机制与总结 270](#_Toc6602573)

[9.3.1 垃圾回收算法 270](#_Toc6602574)

[9.3.2 垃圾回收机制 274](#_Toc6602575)

[9.3.3 垃圾回收有两种类型：Minor GC 和 Full GC。 275](#_Toc6602576)

[9.3.4 垃圾回收算法总结 275](#_Toc6602577)

[9.4 **7种JVM垃圾收集器特点，优劣势、及使用场景** 276](#_Toc6602578)

[9.4.1 直通BAT之JVM系列 276](#_Toc6602579)

[9.4.2 常见的垃圾收集器有3类： 276](#_Toc6602580)

[9.4.3 新生代垃圾收集器 278](#_Toc6602581)

[9.4.4 老年代垃圾收集器 279](#_Toc6602582)

[9.4.5 新生代和老年代垃圾收集器 281](#_Toc6602583)

[9.4.6 JVM垃圾收集器总结 282](#_Toc6602584)

[9.5 **JVM的内存分配、运行原理、回收算法机制** 283](#_Toc6602585)

[**9.5.1** JVM内存模型 283](#_Toc6602586)

[9.5.2 JVM垃圾回收算法 285](#_Toc6602587)

[9.5.3 垃圾回收器 289](#_Toc6602588)

[9.5.4 与GC相关的常用参数 291](#_Toc6602589)

[十、 构架师晋级 292](#_Toc6602590)

[10.1 架构师进阶之独孤九剑：设计模式详解 292](#_Toc6602591)

[10.1.1 为什么要使用设计模式 293](#_Toc6602592)

[10.1.2 设计模式分类和作用 293](#_Toc6602593)

[10.1.3 设计模式书籍推荐 298](#_Toc6602594)

[10.2 **UML是什么?UML常用图以及建模工具有哪些?** 303](#_Toc6602595)

[10.2.1 UML常用建模图 303](#_Toc6602596)

[10.2.2 UML建模时常见的9种图： 304](#_Toc6602597)

[10.2.3 UML工具 306](#_Toc6602598)

[10.3 从程序员进阶到架构师，6大核心技能要领详解 308](#_Toc6602599)

[10.3.1 数据结构和算法 308](#_Toc6602600)

[10.3.2 java语言高级特性 309](#_Toc6602601)

[10.3.3 java web开发 313](#_Toc6602602)

[10.3.4 数据库 315](#_Toc6602603)

[10.3.5 java框架与工具 316](#_Toc6602604)

[10.3.6 大型网站架构设计 318](#_Toc6602605)

[10.4 java架构师进阶之独孤九剑：数据结构以及书籍推荐 319](#_Toc6602606)

[10.4.1 为什么要学习数据结构 320](#_Toc6602607)

[10.4.2 常用的数据结构 321](#_Toc6602608)

[10.4.3 数据结构和算法书籍推荐 324](#_Toc6602609)

[10.5 java架构师进阶之独孤九剑(一)-算法思想与经典算法 326](#_Toc6602610)

[10.5.1 算法思想 327](#_Toc6602611)

[10.5.2 经典算法 330](#_Toc6602612)

[10.6 **java架构师进阶独孤九剑系列（一）：数据结构与算法** 332](#_Toc6602613)

[10.6.1 完整的架构师进阶步骤 333](#_Toc6602614)

[10.6.2 架构师进阶系列之独孤九剑九大步骤： 334](#_Toc6602615)

[10.7 如何学习获得高并发的经验？ 336](#_Toc6602616)

[10.7.1 什么是高并发 336](#_Toc6602617)

[10.7.2 高并发解决方案 337](#_Toc6602618)

[10.7.3 高并发学习经验 339](#_Toc6602619)

[10.7.4 构架师晋级必读书单 339](#_Toc6602620)

[10.7.5 史上最全java架构师技能图谱（上） 354](#_Toc6602621)

[10.8 java程序员的9款开发利器：开发环境+性能调优+代码调试+持续集成工具 373](#_Toc6602622)

[10.8.1 开发环境： 373](#_Toc6602623)

[10.8.2 性能调优 374](#_Toc6602624)

[10.8.3 代码调试 376](#_Toc6602625)

[10.8.4 持续集成:jenkins 378](#_Toc6602626)

[10.8.5 Sonatype Nexus 搭建maven私服 379](#_Toc6602627)

# BAT薪资、技能要求、开发流程

## 岗位、薪资、建议

### 百度

**百度的技术级别**

百度有4万人，**每年招聘应届生技术产品人员1000人左右**，技术岗位级别和阿里类似，分为T序列12级，不赘述了，大概阿里的级别减1或2，就是百度的级别。主要集中T5/T6，升T7很困难，T7升T8更困难；T7以上一般就不做coding了。

一般来说，在百度待3年能给到T5，很多人都等不到三年，原因下面说，社招过来的，一般是外面公司的技术骨干了。T10是技术总监，十个左右；T11是首席科学家。

[**百度薪资**](http://youzhixueyuan.com/tag/%e7%99%be%e5%ba%a6%e8%96%aa%e8%b5%84)**的大概范围**



[百度薪资](http://youzhixueyuan.com/tag/%e7%99%be%e5%ba%a6%e8%96%aa%e8%b5%84)结构：月薪×14.6（12+0.6+2），其他岗位月薪×14，T5以上为关键岗位，另外有股票、期权T5、T6占比最大的级别，T8、T9占比最小级别越高，每档之间的宽幅越大。

### 阿里

**阿里内部的常见级别和定义**

P序列=技术岗 M序列=管理岗



阿里的非管理岗分为10级，其中**P6、P7、P8需求量最大**，也是阿里占比最大的级别。

2

**阿里的级别对应薪资**

阿里薪资结构：一般是12+1+3=16薪，年底的奖金为0-6个月薪资，90%人可拿到3个月。除了薪水，股票也是重要的收入之一，阿里的股票分年限行权，一般**4年拿完**。股票是公司用来奖励员工忠诚度的，所以阿里分年限行权，要离职套现得准备好充足的现金。

### 腾讯

**腾讯的技术级别**

腾讯的分级和阿里/百度都不一样，分为T1/T2/T3/T4共4级，每级又细分为3小级。

员工仍然集中在中段，尤其是 T2.3 和 T3.1

想从T2跨到T3，即从2.3升3.1是非常困难的

### 腾讯的薪酬和级别的关系



### BAT招聘渠道

**应届生入职BAT:校招渠道**

校招的流程。一般提前一年就开始了，需要应届生留意下BAT的招聘官网和官方微博。校招是由集团统一管理的，所以各个事业部不能随时招聘应届生和实习生。选择内推。最好找靠谱的之前在BAT待的师兄和内部同事进行内部推荐，内推的好处是可以选择应聘部门，面试节奏也快点。

校招的要求。BAT倾向于招聘TOP院校的TOP学生，并且专业对口，当然如果能力非常优秀也肯定能进来的。能力上的要求有三点，优秀的基础，很好的学习能力和有自己的作品，一般情况下，这三点满足两点就能通过面试。面试流程是简历筛选+电话初面+电话终面。

**社招进入BAT渠道**

社招的流程，你几乎每时每刻都可以参加BAT的社招，只是有些部门可能有head count的限制，在那个时间点可能不能应聘该部门。

**我以阿里举个栗子。**

一般阿里**P6的社招面试流程是电话面试（异地两次）+现场初面+现场复面+HR面试**，如果级别越高面试流程会越多，基本规律是每高一级多两个人面试。

**阿里技术要求**



如果你是社招，阿里非常需要P6以上的人才：

**阿里P6:能独立解决问题，制定系统的技术实现方案，考核会涉及部分技术深度**。

技能要求：基本的开发相关的算法、JVM、性能调优、框架熟练程度考核。。一定会面试你，还会涉及到架构设计能力、运维的面试。

P7（技术专家）、P8(高级技术专家)。。往上对架构的考核就越严格，之前对知识的掌握是熟练，那越往上就是从**熟练到精通**。

### 我的建议

很多人问我想进入BAT，特别是java系的想进入阿里。。我想说的是，BAT招聘主要就分为两类：校招和社招，校招的硬性要求非常严格，如果你的文凭不行，就好好工作把知识基础打牢了，通过2-3年的项目实践，再通过社招进入BAT，社招也首先推荐内推，内推不行，其实也可以通过优化简历关键词、以及官方投递渠道、进入相关的社群。。总之方法很多，只要你真有本事，工作几年进入不难，难的是你自己平时的技术功底是否扎实。

one more，其实现在TMD等公司需求量非常大，非常渴望技术型人才，特别是今日头条。。，发展非常迅速，对人才的渴望度非常高，刚刚又在上海设立了新的研发团队(非常需要人)。。进入这样的快速发展公司，也是非常不错的选择。

## 阿里P系列技术要求(P5-P7)

**阿里p系列薪资(最新数据比这个高1倍左右)**



### **阿里P5(高级研发工程师)**

**工作要求：**

能独立完成日常工作，并能够对一些方案提出自己的建议。 基本考核就是能上手独立完成工作，熟练掌握。

**技能要求(熟练)：**

   1、程序设计和开发。

    2、工作中需用到的Java开发的技术点以及相关的其他技术点。

    3、工作中需用到的Web开发的技术点以及相关的其他技术点。

    4、工作中需要用到的数据库技术点以及相关的其他技术点。

    5、运维能力

    应用发布和运维，线上故障分析和排查

1-5的技能能力：程序设计和开发、java web、数据库的技能要求请参考我写的这篇文章：[java程序员学习完整线路以及我的学习经验！](http://youzhixueyuan.com/programmers-advance-to-architect-1.html)

### **阿里P6(技术主管)**

**工作要求：**

能独立解决问题，制定系统的**技术实现方案，**考核会涉及部分技术深度。

**技能要求：**

除P5的技能外，还需掌握以下技能，以下技能至少需要做到**熟悉**

    1、架构设计  
    2、运维能力  
    3、操作系统

### **阿里P7(技术专家)**

**工作要求：**

能**独立带领产品或业务向前发展**，并在**产品或业务的规划上**具备一定的能力。

这里我强调一句，是独立带领的能力，并且有一定的产品和业务的规划考核，比如：对阿里的业务是否了解，典型的电子商务的业务模块组成包含哪些等，这一条很多人会忽视业务的重要性，坦率的讲，进入阿里后想晋升，你的业务性能达到，你的上升通道以及空间是非常大的。

**技能要求**：

在P6的基础上，需要在Java开发、Web开发上至少有一项能做到精通。

在架构设计上至少在某一架构领域达到精通。

架构相关的技能要求请参考我写的这篇文章之**大型网站架构设计**篇：

[从程序员进阶到架构师，6大核心技能详解](http://youzhixueyuan.com/the-most-complete-java-architect-skills-map.html)

## 产品技术人必备干货：产品开发流程（完整版）

产品技术开发流程概述

MRD需求分析-原型设计（交互设计）-视觉设计（UI设计）-prd需求文档-研发阶段-测试阶段-上线反馈（效果）

产品技术开发流程

“

产品开发流程涉及的人员从[产品经理](http://youzhixueyuan.com/tag/%e4%ba%a7%e5%93%81%e7%bb%8f%e7%90%86)到设计师、前端、后端等等一系列人员，这篇文章主要关于产品开发的完整流程，希望对产品技术人员有一定的参考价值。

### 产品概念设计：MRD

**产品开发流程步骤第一步**：市场需求文档，简称为MRD（Market Requirement Document），是市场部门的[产品经理](http://youzhixueyuan.com/tag/%e4%ba%a7%e5%93%81%e7%bb%8f%e7%90%86)或者市场经理编写的一个产品的说明需求的文档,该文档是产品项目由“准备”阶段进入到“实施”阶段的第一文档。



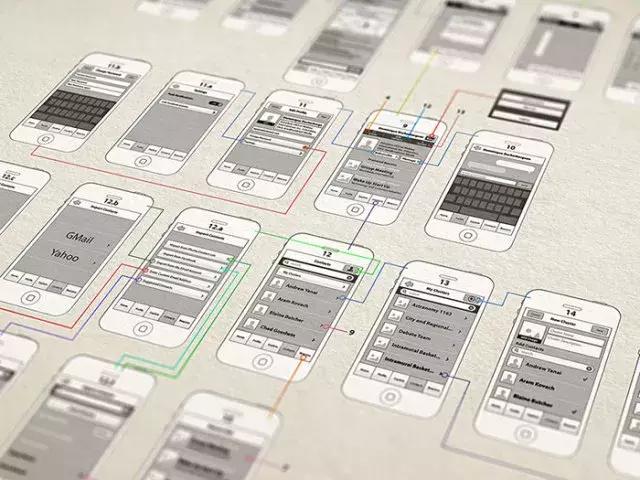
市场需求文档

做任何东西之前，首先要考虑其背后的用户需求、商业价值、技术难度。只有用户有需求，你的产品才会有人用；只有其商业价值成立，才能为企业带来利润，毕竟企业最最基本的目标就是要盈利；只有技术上的总体评估是可行的，整个项目才可被执行。

这个阶段，需要有严格的评审，需要有经验的市场人员、运营人员、产品、技术、测试人员参与，从市场商业和各自专业的角度来参与评审，比如：市场运营人员一般会从市场的角度，产品人员会从用户的角度，技术人员会考虑技术的专业角度来参与评审。

市场需求文档撰写，一般采用ppt xmind keynote的形式，把从市场和用户收集到的数据，以及行业相关的信息进行整理，并且提出对应的需求，大部分的公司都是[产品经理](http://youzhixueyuan.com/tag/%e4%ba%a7%e5%93%81%e7%bb%8f%e7%90%86)来负责MRD和PRD的撰写。

### 二：原型设计(交互设计)



原型设计

原型设计就是交互设计，我比较喜欢用原型设计来表述，主要就是交互设计师与PD、PM、网站开发工程师沟通的最好工具。

原型是一种让用户提前体验产品、交流设计构想、展示复杂系统的方式。就本质而言，原型是一种沟通工具。

**做原型的一般步骤：**



原型设计流程

1：画草图

画草图的目标是提炼想法，画草图要避免陷入审美细节，尽可能快速的导出想法才是关键。

2：演示及讨论

演示和讨论的目标是把一些想法拿出来跟大家分享，然后进一步完善想法。

3：做原型

在明确了想法之后，就可以开始进行原型设计了。这个阶段需要考虑很多细节，找出切实可行的方案，运用合适的原型来表达。

4：测试

原型的目标之一是让受众来检验产品是否达到了预期。

**常用原型设计工具**

Axure、Omnigraffle（Mac）、JustinMind

POP（Prototyping on Paper）、国产Mockplus、Pencil Project 、Proto.io等。

比如：Axure，[产品设计](http://youzhixueyuan.com/tag/%e4%ba%a7%e5%93%81%e8%ae%be%e8%ae%a1)原型最常用的工具，在PC时代就非常出名，主要优点就是上手快，行业内都用，团队协作很方便。

**移动原型设计工具，我经常使用的移动原型设计工具JustMind**

**JustMind**



justinmind

JustinMind是由西班牙JustinMind公司出品的原型制作工具，可以输出Html页面。与目前主流的交互设计工具axure，Balsamiq Mockups等相比，**Justinmind Prototyper更为专属于设计移动终端上app应用。**

这里就不一一说明各原型设计工具，工具只是更加直观的表达你的想法，通过流程的完整演示你的产品，最重要的是你的产品的价值，这一点是要非常明确的，工具只是形式，只要掌握1-2款自己所熟悉的就可以了，**我个人比较常用的是Axure、JustMind，然后再辅助xmind和edraw、visio来画功能框架和流程图。**

### 三： 视觉设计

交互设计一旦通过，就可以进入下一步的高保真图的设计，视觉设计组会根据[产品经理](http://youzhixueyuan.com/tag/%e4%ba%a7%e5%93%81%e7%bb%8f%e7%90%86)的描述，设计出最新的视觉效果(高保真图)，一般采用ps等设计软件。

**工具**

PS(Photoshop) AI(Adobe illustrator) Sketch

**产出**

在交互评审通过后，负责把控UI整体风格和所有视觉效果的设计，主要输出：VI选择的方案、所有视觉效果图、资源包等。

### 四：PRD详细需求文档



prd详细需求文档

PRD详细需求文档一般与视觉设计是同步进行，主要是细化MRD里的功能以及详细流程、文案等细节，主要还是[产品经理](http://youzhixueyuan.com/tag/%e4%ba%a7%e5%93%81%e7%bb%8f%e7%90%86)负责。

PRD详细需求文档，一般包括功能流程图 、产品的功能点(框架脑图)、主要功能卖点，模块的内容，文字文案、各功能描述、弹出框内容文案，各模块之间关系，以及各模块业务逻辑、结构框架图等。

很多[产品经理](http://youzhixueyuan.com/tag/%e4%ba%a7%e5%93%81%e7%bb%8f%e7%90%86)不太重视功能流程图，经常交互到研发和测试的需求文档经常不体现出来，这点是不对的，特别是涉及到负责的流程，需要完整的流程图，需要按照模块的流程逐个进行分类，分类以后按照功能的不同、简易程度，分别画出流程。

### 五：研发阶段

进入研发阶段，主要会涉及到前段开发工程师、架构师、后端工程师、项目经理、测试等人员。



研发流程步骤

**前端工程师**

负责把视觉设计后的高保真图，转换为html、css，利用js实现其交互效果。大家熟知的前段工程师。

**架构师**

一般产品进入研发阶段，大的项目需要做技术的概要设计和详细设计，保证技术方案的稳定性、可扩展性、性能等多项指标，这里就会涉及到架构师的参与,确保技术方案的可行性。

**后端工程师**

实现后端功能开发，让功能可用、易用，「程序猿」是最直接的描述，写代码的牛人或者普通人。

**测试工程师**

测试工程师，会测试产品的缺陷，在各个终端的适应性，以及产品在高访问下的性能测试。

**运维工程师**

测试通过后上线，会涉及到线上硬件运维。

**项目管理**

协调技术、产品、运营等部门把产品更好的实现。



研发流程步骤

一般的研发步骤都会经历如上阶段，概要设计和详细设计主要是对技术的可实施性进行完整的评估，评审通过之后才会进入完整的编码研发和测试阶段，最后才是运维上线，最后才是产品上线后的效果反馈(数据收集验证产品效果)。

# BAT求职

## 5步教你成功求职进入BAT

求职攻略五步曲

填写简历->选择应聘渠道->准备面试->参加面试->选择offer。

### 如何写简历

**一 什么是好的简历**

简历是突出自我亮点的工作或学习经历和自我介绍。所以写简历要把握两个重点，工作经历要突出重点，自我介绍要写全。

**二 如何写简历**

**1：简历的内容**

**简历至少应该包含以下几部分：**

**基本信息**：必填项有姓名，年龄，是否已婚，手机号，常用邮箱，目前所在地，工作经验。目前所在地关系到是进行电话，视频面试还是现场面试。选填项有博客，个人完整，github，英语读写听能力。

**应聘信息**：应聘职位，求职地，职位和求职地都可以写多个。

**自我介绍**：用事实介绍自己的亮点和擅长技能。

**教育背景**：毕业院校，毕业时间。

**工作经验**（由近至远）：在哪些公司任职过，任职时间。好的公司会让面试官有好感。大学生的实习经验也算工作经验。

**项目经验**（由近至远）：写亮点项目，或最能体现自己技术能力的项目，不要把所有项目经验都写上，项目经验是可以引导面试官问问题的方向。

获奖信息和证书：写最能提现自己能力的获奖信息，写多了也没用。证书比如CCNA，CCIE等。

**2：自我介绍**

自我介绍是简历中最重要的。自我介绍要用事实描述，不要用观点描述，比如我有10年Java开发经验，这个是事实，我崇尚团队合作，学习能力强，这个是观点。事实可以证明，而观点很难，所以事实描述比观点描述更有说服力。自我介绍主要是对自己经历的一个亮点总结，比如张三有十年的Java开发经验，其中有两年的Linux操作经验，两年的项目管理经验。张三有自己的个人技术博客XXX.com，经常翻译国内网文章并在XX网站发表，主导或参与XX开源项目。曾经在XX比赛中拿过三等奖。当然类似于学习能力很强的观点描述也可以写，但是要举例子，比如曾经在2周内学习完IOS，并开发了XX应用。

**3: 简历的大小**

简历至少要写两页。举个有趣的事例，我之前公司的同事在面试前筛选简历的办法是，10K以下的简历直接不看，10K的简历一般是一页左右，原因是他认为应聘这么重要的事情，连简历都不愿意好好写的人，估计能力也不行。

**4：简历的格式**

简历一定要美观和简洁，不要让面试官一眼看过去全是密密麻麻的字。

提供PDF或WORD版本的简历，不要提供txt，HTML，ppt格式的简历。

简历的标题。如果你是社招生，写姓名简历\_XX经验\_应聘XX岗位。如果你是应届生，写姓名+XX大学+应聘XX岗位。

我曾经见过最让我震撼的简历是用PPT写的，然后导出成PDF，看得出这个应聘者很用心在写简历。

**三：建议**

期望薪水不要写。因为薪水的计算很复杂，期望薪水是税前还是税后呢，是否包含年终奖呢，是否包含其他个人所得奖金呢。我的建议是根据面试状况最后再谈期望薪水。写上期望薪水万一薪水比面试官还高，面试官可能拿自己的能力来要求你。

不要频繁跳槽。每年换一次工作的求职者简历会不好看，面试官会认为他的职业规划想不清楚，招进来后可能很快又会跳槽。

### 如何选择求职渠道？

**进入BAT有两种途径，校招和社招。**下面我会分别介绍下校招和社招的流程和基本要求，最后分享几个面试小窍门。

**校招的流程**。一般提前一年就开始了，需要应届生留意下**BAT的招聘官网和官方微博**。校招是由集团统一管理的，所以各个事业部不能随时招聘应届生和实习生。选择内推。最好找靠谱的师兄和内部同事进行内部推荐，内推的好处是可以选择应聘部门，面试节奏也快点。

**校招的要求。**BAT倾向于招聘**TOP院校的TOP学生**，**并且专业对口**，当然如果能力非常优秀也肯定能进来的。能力上的要求有三点，**优秀的基础，很好的学习能力和有自己的作品**，一般情况下，这三点满足两点就能通过面试。面试流程是简历筛选+电话初面+电话终面。

**社招的流程**。你几乎每时每刻都可以参加BAT的社招，只是有些部门可能有head count的限制，在那个时间点可能不能应聘该部门。一般P6的社招面试流程是电话面试（异地两次）+现场初面+现场复面+HR面试，如果级别越高面试流程会越多，基本规律是每高一级多两个人面试。

**社招的要求也比较高，倾向于招年轻的最优秀的人才**。最基本的要求是能够独挡一面，来了就干活，不需要太多培训成本。为什么要年轻？年轻意味着有激情，不安于现状，会尝试影响周围的人。为什么招聘最优秀的人？乔布斯说在软件行业优秀的人和普通的人能力上差距可能是两百倍，传统企业没有那么大差距。所以BAT的招聘原则是越优秀越好，可要可不要的不要。有激情的优秀人才会创造无限大的价值。

**社招选择年底应聘**。年底跳槽的人少，各部门的head count必须用完，所以要求可能会相对低些。所以年底相对于年初和年中比较容易进入BAT。

**做好面试准备**。面试前准备一个自己做的最能体现技术能力的事或项目。了解BAT价值观和应聘部门的业务。

利用初面官通过复面。在初面快结束时，面试官出于礼貌会问你有什么想问我的吗？这个时候很多人都不问或者问不关键的问题。我建议询问面试官自己的不足，针对这些不足如何提高，以及自己最得意的项目有哪些不足，如何改进，拿着这些建议和准备去参加复面会更容易通过。

准备好了再投简历。不要乱投简历，我看有的人都有30次投递。因为面试不过你的面试记录会记录下来。如果有很差的评价，以后简历很容易评估不过，都不能进入电话面试。

选择内部推荐。自己投简历可能进入公海，可能不会被人发现。

**如何准备面试**

如果你收到某公司的面试邀请，那就应该开始准备面试，打一场有把握的仗。

**需要准备几件事**，自我介绍，亮点技术或产品，职业规划，这几件事情并不是临时抱佛脚就能完成的，而是要靠平时在工作中的积累，面试进更好的公司只是一个水到渠成的过程，能力到了自然就能进。

准备多久？当时带着强烈想进入BAT的动力，电面准备了几周，复面准备了一个多月。

**第一关，自我介绍**

大部分面试官一般都会让应聘者首先做个自我介绍，自我介绍回合是应聘者最主动的一个回合，因为在这个回合主要是应聘者说，面试官听，所以我认为这个回合非常重要，如果介绍的非常好，不仅可以为后面面试官提问做伏笔，还可以让面试官对你有好感。所以建议准备一个十分钟时长的介绍，我听过很多应聘者的自我介绍一分钟都不到。

自我介绍首先描述自己的基本情况，其次是描述自己的技术亮点，做过的亮点项目或产品。如果没有做过有技术亮点的事，每天都在做增删改查功能或重复性的工作，需要好好反思下，这样下去技术上没有多少增长。如果工作中就是做这个怎么办？可以考虑利用业余时间参与开源项目或自己做一些工具或框架。

**第二关，介绍做过的产品。**

介绍产品时面试官会考察应聘者的沟通能力和思考能力，我们大部分情况都是做产品的一个功能或一个模块，但是即使是这样，自己有没有把整个系统架构或产品搞清楚，并能介绍清楚，为什么做这个系统？这个系统的价值是什么？这个系统有哪些功能？优缺点有哪些？如果让你重新设计这个系统你会如何设计？

**第三关，技术面试**

技术面试主要考察一个人的技术能力，沟通能力和学习能力。不同的面试官的问题不一样，没有规律可循，所以首先在网上搜下该公司主要的面试题，必考的肯定是基础，好的面试官会考察你擅长的技术，看看你是否有很强的学习能力和技术能力。有的面试官会问自己擅长的技术，这样对面试者的要求就更高，所以你要准备的更加充分，在技术的宽度上学习下。深度优先，广度次之。

**最后一关，HR面试**

HR面试主要考察一个人的价值观，潜力和职业规划。所以进入这一关之前请想清楚几个问题。

为什么想加入这家公司？你需要了解下这家公司的文化和氛围。

为什么离开上一家公司？钱少，不能实现梦想，还是有哪些不爽的地方。如果是换工作为了实现梦想，那么你的梦想是什么。不爽的地方如果在新公司再遇到怎么处理。

你的职业规划是什么？

当你遇到瓶颈的时候你怎么办？

这些年你每年的成长是什么？怎么能体现你的进步？

### 如何现场面试?



可以参考之前写的这篇文章“

**一位前BAT面试官详谈进入**[**BAT面试经验**](http://youzhixueyuan.com/tag/bat%e9%9d%a2%e8%af%95%e7%bb%8f%e9%aa%8c)

### 如何选择Offer?

之前很多应届生咨询我如何选择offer，某某公司的面试我是否应该参加，某某公司的Offer我是否应该拒绝？

大家困惑的地方有几点：

**第一是薪资问题。**

**第二如何选择公司。**

**第三如何选择行业。**

**薪资问题**

刚毕业几年不要看薪资，除非薪资的差距是在数量级间的差距，如果只是几千块的差距这个不算差距，现在的工资并不代表你未来的工资，学会投资自己看未来，成长性好的员工未来的收益差距是在数量级，比如几年后，同学A的薪水比另外一个同学B的薪水高上百万都是非常正常的。

其次要学会比较薪水。收到Offer时，首先要知道薪酬福利的组成，比如月薪，年终奖，期权，商业保险，补贴等。不要单纯的比较月薪，而是综合比较年薪和福利。A同学月薪比B低几千，但是A同学年薪和福利加在一起可能比B同学高好几倍。除了期权外，福利上主要关注以下几点：

公积金，等你买房或退休的时候可以取出来，这个我认为可以算在薪水里。国家规定公司和个人都要缴纳12%，但是有些企业缴纳的公积金只有8%，因为他们只缴纳基本工资的12%，比如我曾就职的一家企业的工资=基本工资+岗位工资+绩效工资，基本工资只有65%，然后按照65%的工资缴纳%12的公积金，这样无形中比同样月薪的人低了4%的收入。

补贴。不同的公司补贴不一样，大致有住房补贴，异地补贴，汽油补贴，餐补等。如果你从A城市去B城市上班有迁移成本，可以咨询下公司是否有住房补贴。另外补贴有个期限，是一年还是几年，这个也要问清楚。

商业保险，过节费等。

### 如何选择公司

我自己也经历过几个不同类型的公司，小型私企，大型私企，互联网企业。从我的经历来看，如果你想学技术可以选择互联网公司，外企和创业公司，如果你想做管理，可以选择创业公司和中大型私企。

我总结了一下各种类型公司的状态，仅供参考：

以上内容大部分都是帮助进入BAT等一线互联网公司的方法。成长还有很多别的方法和渠道。在创业公司你一样可以锻炼的很出色。但是按照几率来讲，早期进入一线互联网企业，更有利于未来的发展，特别是做技术开始的。

**最后，除了以上部分，谈谈我个人的建议，仅供参考：**

任何是事情要学会先思而行，不仅仅适用于工作选择，更适用于发展。没想好，没想透不要紧，但一定要有这样的思维方式。

以后选择可以加入一条，行业选择。做技术的朋友没有过多考虑行业的重要性，因为去一家公司更多是实现，这是一个极大的误区。要开始学会行业的纵深发展。再过几年也许，你会有更深的体会。

## 手把手教你求职进入BAT

### 求职跳槽的三大误区



**第一个误区，求职跳槽就是找工作之前3个月的事。**

如果想进入自己心仪的公司，首先肯定需要方法和长时间准备的。而且，越好的公司，准备的时间越长，准备跳槽的时间周期，最好以年为单位，而不是3月的临时备战。

**第二个误区，以纯生态的方式去面试，简历不做优化，谈吐技巧不做改进，以原味出现**。

很多人会说，这不是简历作弊吗？请记住，这并不是让你去瞎改莫需有的经历和职位，这是优化你自己。人靠衣装，马靠鞍，简历就是每一个人的名片和身份，一定要更早的意识到并“优化”自己，而不是有意去“破坏”。现在，基本每一份简历，在你投递以后，都有第三方的公司，系统结合人工的方式，鉴别简历真假，只不过你不知道而已。所以，很多时候你的简历已经进入了大部分公司的黑名单，你还不知道是为什么。你能想到的，其实公司在面试的第一个环节已经通过第三方公司做过筛选了。

**第三个误区，我的履历是小公司，想进大公司需要镀金。**

很多小公司的朋友，一直比较懊恼的事就是没有大公司的经历和背书。一直认为这是自己不能进入大公司的原因。其实，小公司也有自己的优势，只不过你没有在你的简历上重点突出而已。其实，这些优势也是大公司目前找人最看重的，你却没写。稍后，我会给到大家具体的例子，看到小公司去BAT的典型案例。

我见过很多完全能力在BAT等一线互联网公司胜任朋友，去面试过一两次就放弃了，时间一长，这事就完全放弃了。其实，并不是实力不够，主要是方法不当。如果你真打算进去，其实方法真的太多。比如，我以前在淘宝的同事，就是同时投阿里好几个子公司的简历，去面试了阿里多家子公司，最后选择了去淘宝。当然，除了他的硬实力以外，他也有一套没有告知你的方法。

如果把这些误区都正视了，自然就会开始重视和思考怎样才能去自己心仪的公司的方法和步骤。

以下谈谈我之前面试公司的准备步骤和心得，作为一个曾经的面霸，希望对你们有所用。

面霸可不是吹的：比较大的互联网集团公司，比如：阿里的CTO,携程的CTO,搜狐的CTO都面试通过，中间包含多轮的技术硬实力和软实力的面试。

### 第一步：简历自我评价改进

突出目前优势-未来成长优势-独特简历介绍

简历就是每一个人的招牌，招牌必须得帅。

之前谈到的几位读者朋友，我第一眼看到他们简历的时候，基本都是千篇一律。并不是说简历的个人经历，而是简历的排版以及不突出重点，看到的都是千篇一律的简历。这在茫茫人海中，怎样能率先凸显出你的价值呢？

**首先，需要快速改进自己的招牌。**

每一个简历基本都有自我评价这一项，很多朋友并不是很关心，苏不然就是在这个环节丢失了面试的机会。

我作为面试官，不太可能仔细查看完整你的简历，除非你的简历很特别。首先，看的就是你的自我评价。如果在自我评价环节，看到了有价值的信息，才会继续往下看完整简历。

我截取一小段我的简历的自我评价作为例子：

在**互联网电子商务行业和旅游行业**，有着**10年以上**的**产品技术资深的经验**：先后在**美国新蛋newegg、淘宝、百度、携程**等大型互联网公司工作，从最早美国新蛋一名**程序员**，到**淘宝的技术架构**工作，再到**百度的研发经理**，再到现在的**携程定制旅游CTO**，**管理产品技术等团队**，一路的成长轨迹已经成为自己最大的财富。

…………….

后面是你在工作期间取得的**重要的业绩**，比如：在公司期间的最佳团队奖(说明你的管理实力)，这里要结合你面试的公司的职位来突出业绩点。

首先是让对方对你有兴趣，其次才是你取得的成绩，尽可能把自己的优势突出在最前方。

当然，我知道你肯定会说，我没有你这样好的履历。那就拿出你现阶段，最拿得出手的优势点，放在最前面。

举个栗子，你没有大公司的经历，那就突出你自己本身，比如**你的成长经历，或者你的自学成才经历，你的项目历练经历**等。这个阶段没有大公的履历，那就要突出**自我未来成长优势。**

还是举个栗子，俞军，前百度产品副总裁、首席产品架构师，互联网产品经理的祖师爷，我截取一段他当时面试百度的简历中的自我评价。

26岁，上海籍，同济大学化学系五年制，

**览群书，多游历。**

长期想踏入**搜索引擎业**，无奈欲投无门，心下甚急，故有此文。

如有公司想做**最好的中文搜索**，诚意乞一参与机会。

**自我评价：**

本人**热爱搜索成痴**，只要是做搜索，不计较地域**（无论天南海北，刀山火海）**，不计较职位（**无论高低贵贱一线二线，与搜索相关即可**），不计较薪水（**可维持个人当地衣食住行即是底线**），不计较工作强度（反正已习惯了**每日14小时工作制**）。

如果是你在找搜索相关的人，你看到这样一份简历会怎么样，是不是会眼前一亮呢？

这个阶段，你看到有任何大公司履历吗，没有，有的就是自己想励志从事搜索行业的抱负和雄心壮志，这就叫突出**自我未来成长的优势**。

这里还给大家提醒一点，简历内容不要再出现什么**精通、熟悉**等字样，需要把精通、熟悉转换成更具体的内容，并且把简历用ppt等工具重新制作，最好转换成pdf格式。

**简历切记别千篇一律！**

### 第二步：简历项目经验改进



有了第一步自我评价的敲门砖，才会有项目经验的介绍。

很多朋友喜欢把自己所有的项目经历都写一遍，请切记，项目经历仍然是突破自己的优势。

首先，要做的就是精简你的项目经历。

我相信，你一定会有你拿得出手的项目，你认为还不错的项目，重点突出这些项目的详细描述，不重要的经历就一笔带过即可。

还是举一个栗子，我的简历项目描述。

负责携程旅游事业部签证，国际火车票，定制旅游三大业务线的产品和技术团队管理工作。具体工作分为**五个方面**：团队建设和管理，业务的产品规划和把控，负责整体技术架构的规划和把控，项目的推动和持续跟踪，敏捷教练：

1.**团队建设和管理**：组建整个团队，团队成员包含：产品经理，开发主管，开发工程师，项目经理，测试主管，测试人员。

2.**业务的产品规划和把控**：负责各业务线产品的需求调研以及规划，制定各业务板块每一个季度详细产品规划方案，并且跟踪实施和改进；

3**.负责整体技术架构规划和把控**：负责各业务板块的整体技术系统架构设计方案，推动各业务板块往大型分布式系统演进；

4.**项目的推动和持续跟踪**：把控开发任务的需求分析和开发计划制定；带领以及指导整个团队：产品、技术、测试，用敏捷的方式快速推进产品和技术计划；

5.**敏捷教练**：第一个在携程内部，采用敏捷的方式驱动产品技术，帮助成员跟上敏捷的节奏。

……

对应的业绩等，比如我的业绩更多就是业务的KPI含量比例更重，比如，完成了整年的业绩，采用部分技术的自动化代替人工等，为公司节约了具体多少money等。

这样，是否把具体你的工作职责拆解成具体的内容，并这些工作内容，还能突出你的优势呢？

当然，你需要根据你当前的情况来具体分析。比如，这个阶段你是程序员，产品经理等，自我评价和项目经验等，都是一脉相承，突出自己的优势。

### 第三步：定制公司的面试简历



在互联网大数据的背景时代下，谈到的都是千人千面，为什么简历就不能是千人千面呢？

很多朋友投递简历到公司的时候，采用的都是机海战术，认为都是多多益善。

其实，投递简历不需要太多，重点是找到投递的渠道，以及根据公司的情况来改进自己的简历。

举一个栗子，你准备面试架构师 or 项目经理，又 or你打算技术转产品，是不是你投递的简历里，是否应该突出具体从事的工作，与你相关的优势点呢？

这里还给大家提醒一点，这里最好结合你投递的公司来优化你的简历。每一个公司的文化和氛围都不一样，就拿我去过的这几家公司来讲，文化氛围差别都特别大。所以，对应的面试官，查看简历以及现场面试的时候，除了你的硬实力考察以外，软实力的考察还不太一样。这些，你可以考虑在简历根据公司的情况，来重点凸显。

## 如何提高面试成功率

很多人说到面试，头一下就大了，各种不自信，是实力不够还是信心不够？面试除了你平时不断积累实力，另一方面也需要讲究面试方法和技巧。我特别喜欢一句话：不打没有准备的战。

are  you ready?

### **第一步 快速聚焦岗位职责**

凡事讲究方法，多用心、多动脑、多动手。**1.至少职责先匹配上吧**重点关注**面试职位**的**岗位职责**，结合该岗位需要的知识体系，开始准备，这样**聚焦**。知识体系的匹配。很简单，自己上网搜，各种题库、各种解药。

**2.凡事都要有针对性**尽量多了解相关公司所属行业地位和实际情况。根据成熟型公司或者创业型公司，进行适当调配重点。

### **第二步 准备你的亮点**

岗位能匹配了，至少证明你能干活了，关键的来了。**你需要准备你的成功案例**不管你是做技术的、产品、运营，你有一件是你自己特别自豪，特别牛的案例？拿出来，给我们分享下。关于亮点，很多人不以为然，**讲故事是一种很强的本事。**如果你能很有逻辑的把一件事在10分钟内说得很清楚，这难道不是一种很强的沟通能力？**反复讲述你的案例**关于亮点的讲述，方法很简答，反复复盘。最好的方法就是不断练习，练习从头到尾的逻辑性，以及记录下你讲述的案例重点。**再次练习。**

### **第三步 展示你强大的自信**

很多我面试过的人，有部分能力差一点的，最后让他们通过。很重要的一项：你的自信心。**从自我介绍开始，全程保持微笑**微笑的力量非常巨大，谁能拒绝一个爱笑的人呢？微笑，它可以快速缩短你与面试官之间的心理距离，为后续深入沟通，创造了一个和谐的氛围。**眼神不要飘忽，讲述平缓一点**还是聚焦，先聚焦面试官的眼睛珠:)，这是一种最基本的尊重，也是展示你自信的一种方式。**最后，记得与面试官互动**如果你到了这里，请记得为面试官点个赞。如果他认可了你，一定会给你分享他的故事。总之，想提高面试成功率，需要硬实力和软实力的同时提高。

## 前BAT面试官详谈进入BAT面试经验

之前有一篇文章“最强攻略：java程序员如何通过阿里、百度社招面试”谈到了进入[BAT](http://youzhixueyuan.com/tag/bat)需要掌握的技术知识点,关注优知学院可以获得。很多同学看完后，一直在说能否与面试一起结合谈谈这方面的经验。所以，补充了本篇文章。  
希望能使大家在未来少走点弯路！

找工作之前，有一点你必须清楚，就是找工作是一件看缘分的事情，不是你很牛逼，你就一定能进你想进的公司，都是有一个概率在那。如果你基础好，项目经验足，同时准备充分，那么你拿到offer的概率就会比较高；相反，如果你准备不充分，基础也不好，那么你拿到offer的概率就会比较低，但是你可以多投几家公司，这样拿到offer的几率就要大一点，因为你总有运气好的时候。

所以，不要惧怕面试，刚开始失败了没什么的，多投多尝试，面多了你就自然能成面霸了。得失心也不要太重，把每次面试都当做一次学习经验。

以下就是我个人总结的部分[面试经验](http://youzhixueyuan.com/tag/%e9%9d%a2%e8%af%95%e7%bb%8f%e9%aa%8c)，其中有我面试别人的，也包含早期我个人面试的经验回想。



**整个面试过程，无非就上面四个环节。**

### 1.从自我介绍开始，全程保持微笑

微笑的力量非常巨大，谁能拒绝一个爱笑的人呢？微笑，它可以快速缩短你与面试官之间的心理距离，为后续深入沟通，**创造了一个和谐的氛围**。

个人履历，尽量简短，重点讲自己的亮点。比如，之前在哪家公司有实习、或者哪家公司做了什么你觉得还不错的项目。自己的履历中关于城市、学校等可以简短描述，突破出重点为主。

在介绍自己的同时，眼神不要飘忽，眼神需要专注，讲述语气可以稍微平缓一点。这是一种最基本的尊重，也是展示强大你自信的一种方式。

第一环节很重要，面试如果对你留下了较好的印象：**主动、阳光、自信**，对你后面的面试环节都是加分选项。

备注：[BAT](http://youzhixueyuan.com/tag/bat)最喜欢这样有活力，有动力，又有主动性的年轻人。一般都是重点培养，甚至技术环节差点，都可以通过。

所以，千万别轻易别低估这个环节。有部分 同学不注意这个环节，中间部分小环节出了偏差，甚至让面试官直接否决了。原因很简单，除了招你进来，他也要考虑培养，以及团队执行等问题。谁愿意招一个自以为是的人在自己的团队呢，哪怕你技术可以又怎样，面试官没有理由非招你莫可。

所以，以上备注内容才是重点。

### 2.技术面试

这就是考你的技术基本功了，进入[BAT](http://youzhixueyuan.com/tag/bat)一般问题的技术问题都这些，就看你的掌握程度了。

如果你还想了解具体技术面试题目，可以看我之前的这篇文章“最强攻略：java程序员如何通过阿里、百度社招面试”，里面基本都是我在[BAT](http://youzhixueyuan.com/tag/bat)面试别人以及早期参加[BAT](http://youzhixueyuan.com/tag/bat)面试环节，总结出来的[BAT](http://youzhixueyuan.com/tag/bat)必问技术题目内容。

备注：有很多刚入道不久的同学，总想有什么捷径！捷径就是：掌握这些内容，不要只为了面试而面试。你的工作早晚也需要用到这些知识点。

### 3.项目介绍

如果以上两个环节，你都通过了，自然面试官就会问你最擅长的项目是什么？

你认为你做过有技术含量的项目是什么？

看看你的项目实力、看看你的描述项目的语言组织逻辑、特否快速抓住重点描述。

在参与这个项目的过程中，你扮演的角色是什么？

如果这个项目真不错，看看你在项目中的重要性。客观，得出你的部分技术实力。

整个项目，有什么挑战的技术难点，能否画一个技术架构图、部署图等等？

把整个项目的技术难度表现出来，单从技术的角度视野来评估你。

### 4.职业规划

当然，如果以上环节你都通过了。到这一步，基本就给中国好声音一样，“你的梦想是什么”？

谈谈你的职业规划吧。

请切记，这个环节还真不是中国好声音，导师准备录用你了，问你的梦想，这里其实有很多坑。很多面试者忽视了这个环节的重要性，特别是到了后面，更要谨慎，需要认真对待这个环节。

面试官其实还在继续考察你。如果你面试的是纯粹技术岗位，直接说以后想做技术架构师就成了。千万别画蛇添足，谈一堆给技术不相关的事，什么管理啊等等，请记住，这是大忌！什么阶段干什么样的事，你有梦想也好，以后再详谈不迟。

以上，就是我在BAT期间面试别人以及自己总结的早期面试BAT经验。希望能使大家能少走点弯路，早点进入正轨，从而迈向更好的职业生涯。

## 怎样跳槽最有利于自身的发展？

[跳槽](http://youzhixueyuan.com/tag/%e8%b7%b3%e6%a7%bd)不只是纯粹的“跳”，跳前的思考，跳前的准备，跳前的判断从何而来

希望本篇能给你带来一点反思。

### 主动求发展

我把这一类放在第一项，主动出击求发展，这类属于已经有很明确的职业发展目标。

清楚得失，知道自己应该去哪，可以暂时忍受薪资降低。

补充一个例子，我认识一个朋友在深圳，为了叙事方便，暂时称为A先生。A先生是一个家创业公司的技术经理月薪在3万左右，之前做了一个决定，离开技术这个行业，加入一家创业的金融公司做销售。从技术直接转岗到另一个行业的销售，这个跨度很大。据后面A先生描述，前3个月一个单子都没有出，薪资只有底薪，从每月3万到每月1000元，甚至这1000元都快没了。A先生早期面临了很大的压力，为了活下来，A先生给自己定了一个小目标，再干3个月,为自己的最后的自损线，再不行就回去。据A先生描述，刚好他的第4个月出业绩了，后来信心来了，不断改进，越做越好，第5个月做到了他们公司第一。最近听他分享了部分心得：很满意这次转型，从眼见到成熟度，再到个人性格塑造，再到薪资，都有了很大的进展。

当然，不是每一个人都能有这么好的结局，但是，如果你能承受这件事坏的方面，这样的尝试何乐而不为呢。毕竟，为你人生开启了一扇窗，一条更精彩的路。

### 发展遇到瓶颈

技术岗位的发展瓶颈比较多，如果你遇见了瓶颈，请认真思考诸如此类问题，[跳槽](http://youzhixueyuan.com/tag/%e8%b7%b3%e6%a7%bd)后能否解决。

**准备转管理，思维没跟上。**

之前纯做技术岗位，如今准备往管理发展。从最早做好自己，到现在需要帮助别人成就自己，这是第一次思维方式的大转变。

**沟通能力比较差，经常被同事嘲笑情商低。**

任何人经常与机器打交道，疏于人打交道，交流沟通肯定会弱化，这个是不争的事实。接受现状，偶尔自嘲自己，然后再改进，是最好的办法。

**技术在公司没有得到尊重，毫无话语权。**

每周例会，技术部经常被同级部门挤兑，自己毫无办法，甚至老板也这样觉得。其实并不是你不行，而是你让他们觉得你不行。学会发声，你真做好了，谁敢挤兑你，很多技术人觉得发声很难，还是老话，多练习。

**安于现状,一件事情能重复做5年**

一个业务模块做了5年，你说还有个哪个产品经理比你更懂。稍微有点产品意识的，早转产品了，那你还抱怨说你没机会。

### 遇见不靠谱老板

**承若的事情，不兑现**

老板年初的时候，信誓旦旦的说，我们今年完成什么目标，我给你们奖励…,当你完成目标后，最后他拒绝履行若言。

**拒绝承担责任**

按照老板的思路上线了一款产品，上线后效果达不到预期，把责任推卸到产品、技术部门。

**过河拆桥**

早期与公司一起创业走来，不想，老板很快请来空降兵。架空你的权利，各种让你在公司不爽，逼你走。美其名曰：你跟不上公司的发展。

**吝啬**

职位名称越来越好，薪资和待遇不见涨，美其名曰：我给了你机会，你就应该这样回报我。

**喜欢权术**

擅长厚黑学，把员工看成个人资产，毫无尊重而言，每天需要员工揣摩圣意。

**任人唯亲**

不以结果为导向，只凭个人喜好用人。只要你听话，我就用你。不管你做出了多大的成绩，如果你想晋级，抱歉，只要我在，你就不可能。

**自私**

从不从员工角度思考问题，只考虑自己。

**这些都是不靠谱的老板，你所要做的就是远离。**

**最后**

你只有吃透你的[跳槽](http://youzhixueyuan.com/tag/%e8%b7%b3%e6%a7%bd)诉求，再排除部分不靠谱的干扰，我想一定会比上次选择的更好。

# 阿里P8架构师谈：

## HashMap的底层结构、原理、扩容机制

the-underlying-structure-and-principle-of-hashmap.html

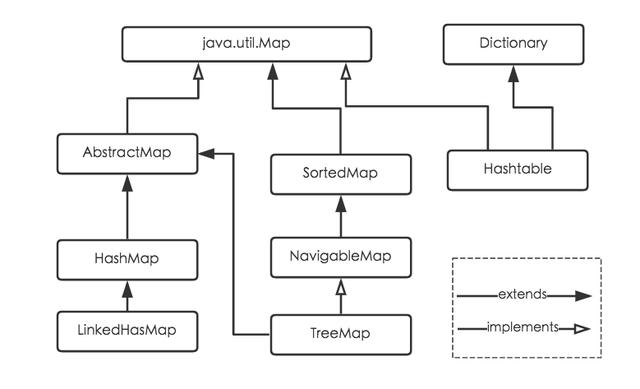
### 摘要

HashMap是Java程序员使用频率最高的用于映射(键值对)处理的数据类型。

随着JDK（Java Developmet Kit）版本的更新，JDK1.8对HashMap底层的实现进行了优化，例如引入红黑树的数据结构和扩容的优化等。本文结合JDK1.7和JDK1.8的区别，深入探讨HashMap的结构实现和功能原理。

### 简介

Java为数据结构中的映射定义了一个接口java.util.Map，此接口主要有四个常用的实现类，分别是HashMap、Hashtable、LinkedHashMap和TreeMap，类继承关系如下图所示：



**下面针对各个实现类的特点做一些说明：**

(1) HashMap：它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。

HashMap最多只允许一条记录的键为null，允许多条记录的值为null。HashMap非线程安全，即任一时刻可以有多个线程同时写HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections的synchronizedMap方法使HashMap具有线程安全的能力，或者使用ConcurrentHashMap。

(2) Hashtable：Hashtable是遗留类，很多映射的常用功能与HashMap类似，不同的是它承自Dictionary类，并且是线程安全的，任一时间只有一个线程能写Hashtable，并发性不如ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁。Hashtable不建议在新代码中使用，不需要线程安全的场合可以用HashMap替换，需要线程安全的场合可以用ConcurrentHashMap替换。

(3) LinkedHashMap：LinkedHashMap是HashMap的一个子类，保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。

(4) TreeMap：TreeMap实现SortedMap接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator遍历TreeMap时，得到的记录是排过序的。如果使用排序的映射，建议使用TreeMap。在使用TreeMap时，key必须实现Comparable接口或者在构造TreeMap传入自定义的Comparator，否则会在运行时抛出java.lang.ClassCastException类型的异常。

对于上述四种Map类型的类，要求映射中的key是不可变对象。不可变对象是该对象在创建后它的哈希值不会被改变。如果对象的哈希值发生变化，Map对象很可能就定位不到映射的位置了。

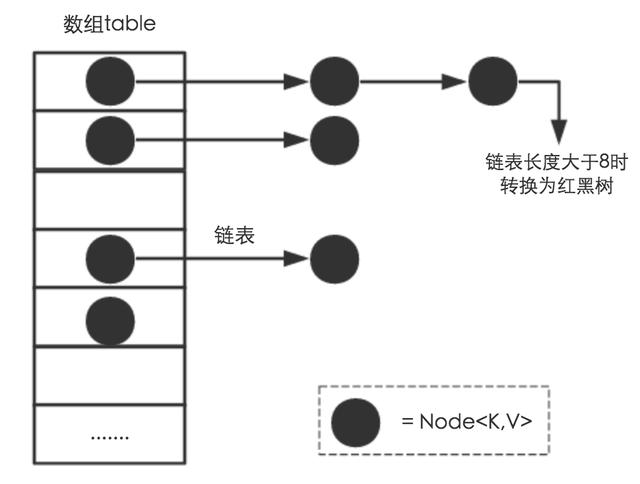
通过上面的比较，我们知道了HashMap是Java的Map家族中一个普通成员，鉴于它可以满足大多数场景的使用条件，所以是使用频度最高的一个。下文我们主要结合源码，从存储结构、常用方法分析、扩容以及安全性等方面深入讲解HashMap的工作原理。

#### 内部实现

搞清楚HashMap，首先需要知道HashMap是什么，即它的存储结构-字段；其次弄明白它能干什么，即它的功能实现-方法。下面我们针对这两个方面详细展开讲解。

##### 存储结构-字段

从结构实现来讲，HashMap是:**数组+链表+红黑树**（JDK1.8增加了红黑树部分）实现的，如下如所示。



**这里需要讲明白两个问题：数据底层具体存储的是什么？这样的存储方式有什么优点呢？**

(1) 从源码可知，HashMap类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个Node的数组。我们来看Node[JDK1.8]是何物。

static class Node<K,V> implements Map.Entry<K,V> {

final int hash; //用来定位数组索引位置

final K key;

V value;

Node<K,V> next; //链表的下一个node

Node(int hash, K key, V value, Node<K,V> next) { ... }

public final K getKey(){ ... }

public final V getValue() { ... }

public final String toString() { ... }

public final int hashCode() { ... }

public final V setValue(V newValue) { ... }

public final boolean equals(Object o) { ... }

}

Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。上图中的每个黑色圆点就是一个Node对象。

(2) HashMap就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java中HashMap采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。例如程序执行下面代码：

map.put("优知","IT进阶站");

系统将调用”优知”这个key的hashCode()方法得到其hashCode 值（该方法适用于每个Java对象），然后再通过Hash算法的后两步运算（**高位运算和取模运算**，下文有介绍）**来定位该键值对的存储位置，有时两个key会定位到相同的位置，表示发生了Hash碰撞**。当然Hash算法计算结果越分散均匀，Hash碰撞的概率就越小，map的存取效率就会越高。

如果哈希桶数组很大，即使较差的Hash算法也会比较分散，如果哈希桶数组数组很小，即使好的Hash算法也会出现较多碰撞，所以就需要在空间成本和时间成本之间权衡，其实就是在根据实际情况确定哈希桶数组的大小，并在此基础上设计好的hash算法减少Hash碰撞。那么通过什么方式来控制map使得Hash碰撞的概率又小，哈希桶数组（Node[] table）占用空间又少呢？答案就是好的Hash算法和扩容机制。

在理解Hash和扩容流程之前，我们得先了解下HashMap的几个字段。从HashMap的默认构造函数源码可知，构造函数就是对下面几个字段进行初始化，源码如下：

int threshold; // 所能容纳的key-value对极限

final float loadFactor; // 负载因子

int modCount;

int size;

首先，Node[] table的**初始化长度length(默认值是16)**，**Load factor为负载因子(默认值是0.75)**，threshold是HashMap所能容纳的最大数据量的Node(键值对)个数。threshold = length \* Load factor。也就是说，在数组定义好长度之后，负载因子越大，所能容纳的键值对个数越多。

结合负载因子的定义公式可知，threshold就是在此Load factor和length(数组长度)对应下允许的最大元素数目，超过这个数目就重新resize(扩容)，扩容后的HashMap容量是之前容量的两倍。默认的负载因子0.75是对空间和时间效率的一个平衡选择，建议大家不要修改，除非在时间和空间比较特殊的情况下，如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值；相反，如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1。

size这个字段其实很好理解，就是HashMap中实际存在的键值对数量。注意和table的长度length、容纳最大键值对数量threshold的区别。而modCount字段主要用来记录HashMap内部结构发生变化的次数，主要用于迭代的快速失败。强调一点，内部结构发生变化指的是结构发生变化，例如put新键值对，但是某个key对应的value值被覆盖不属于结构变化。

在HashMap中，哈希桶数组table的长度length大小必须为2的n次方(一定是合数)，这是一种非常规的设计，常规的设计是把桶的大小设计为素数。

这里存在一个问题，即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响HashMap的性能。于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当**链表长度太长（默认超过8）时，链表就转换为红黑树**，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。本文不再对红黑树展开讨论，想了解更多红黑树数据结构的工作原理。

##### 功能实现-方法

HashMap的内部功能实现很多，本文主要从：

**1).根据key获取哈希桶数组索引位置**

**2).put方法的详细执行**

**3).扩容过程三个具有代表性的点深入展开讲解。**

###### 确定哈希桶数组索引位置

不管增加、删除、查找键值对，定位到哈希桶数组的位置都是很关键的第一步。前面说过HashMap的数据结构是数组和链表的结合，所以我们当然希望这个HashMap里面的元素位置尽量分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，不用遍历链表，大大优化了查询的效率。HashMap定位数组索引位置，直接决定了hash方法的离散性能。先看看源码的实现(方法一+方法二):

方法一：

static final int hash(Object key) { //jdk1.8 & jdk1.7

int h;

// h = key.hashCode() 为第一步 取hashCode值

// h ^ (h >>> 16) 为第二步 高位参与运算

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

方法二：

static int indexFor(int h, int length) { //jdk1.7的源码，jdk1.8没有这个方法，但是实现原理一样的

return h & (length-1); //第三步 取模运算

}

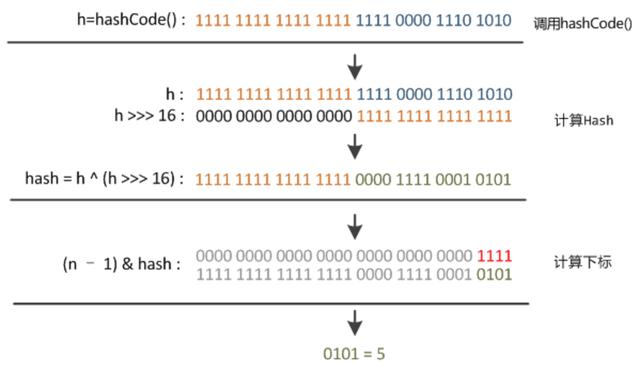
这里的Hash算法本质上就是三步：**取key的hashCode值、高位运算、取模运算**。

对于任意给定的对象，只要它的hashCode()返回值相同，那么程序调用方法一所计算得到的Hash码值总是相同的。我们首先想到的就是把hash值对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，模运算的消耗还是比较大的，在HashMap中是这样做的：调用方法二来计算该对象应该保存在table数组的哪个索引处。

这个方法非常巧妙，它通过h & (table.length -1)来得到该对象的保存位，而HashMap底层数组的长度总是2的n次方，这是HashMap在速度上的优化。当length总是2的n次方时，h& (length-1)运算等价于对length取模，也就是h%length，但是&比%具有更高的效率。

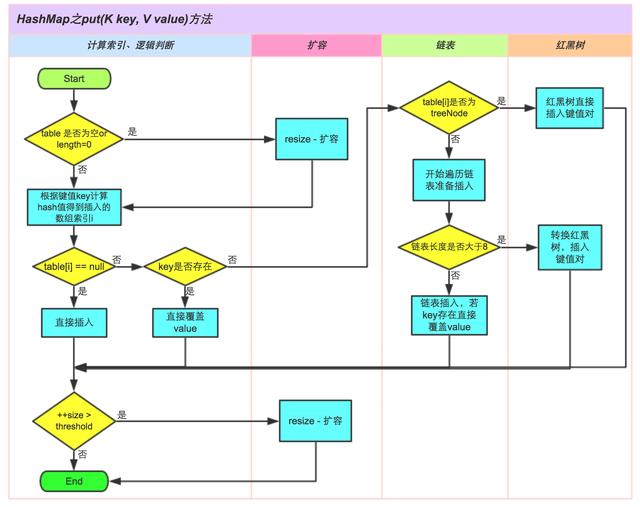
在JDK1.8的实现中，优化了高位运算的算法，通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在数组table的length比较小的时候，也能保证考虑到高低Bit都参与到Hash的计算中，同时不会有太大的开销。

下面举例说明下，n为table的长度。



###### 2. 分析HashMap的put方法

HashMap的put方法执行过程可以通过下图来理解，自己有兴趣可以去对比源码更清楚地研究学习。



①.判断键值对数组table[i]是否为空或为null，否则执行resize()进行扩容；

②.根据键值key计算hash值得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；

③.判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向④，这里的相同指的是hashCode以及equals；

④.判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤.遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

⑥.插入成功后，判断实际存在的键值对数量size是否超多了最大容量threshold，如果超过，进行扩容。

###### 3. 扩容机制

扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然Java里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

我们分析下resize的源码，鉴于JDK1.8融入了红黑树，较复杂，为了便于理解我们仍然使用JDK1.7的代码，好理解一些，本质上区别不大。

1 void resize(int newCapacity) { //传入新的容量

2 Entry[] oldTable = table; //引用扩容前的Entry数组

3 int oldCapacity = oldTable.length;

4 if (oldCapacity == MAXIMUM\_CAPACITY) { //扩容前的数组大小如果已经达到最大(2^30)了

5 threshold = Integer.MAX\_VALUE; //修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了

6 return;

7 }

8

9 Entry[] newTable = new Entry[newCapacity]; //初始化一个新的Entry数组

10 transfer(newTable); //！！将数据转移到新的Entry数组里

11 table = newTable; //HashMap的table属性引用新的Entry数组

12 threshold = (int)(newCapacity \* loadFactor);//修改阈值

13 }

这里就是使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将原有Entry数组的元素拷贝到新的Entry数组里。

1 void transfer(Entry[] newTable) {

2 Entry[] src = table; //src引用了旧的Entry数组

3 int newCapacity = newTable.length;

4 for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组

5 Entry<K,V> e = src[j]; //取得旧Entry数组的每个元素

6 if (e != null) {

7 src[j] = null;//释放旧Entry数组的对象引用（for循环后，旧的Entry数组不再引用任何对象）

8 do {

9 Entry<K,V> next = e.next;

10 int i = indexFor(e.hash, newCapacity); //！！重新计算每个元素在数组中的位置

11 e.next = newTable[i]; //标记[1]

12 newTable[i] = e; //将元素放在数组上

13 e = next; //访问下一个Entry链上的元素

14 } while (e != null);

15 }

16 }

17 }

newTable[i]的引用赋给了e.next，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到Entry链的尾部(如果发生了hash冲突的话），这一点和Jdk1.8有区别。在旧数组中同一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。

### 线程安全性

在多线程使用场景中，应该尽量避免使用线程不安全的HashMap，而使用线程安全的ConcurrentHashMap。

那么为什么说HashMap是线程不安全的，下面举例子说明在并发的多线程使用场景中使用HashMap可能造成死循环。代码例子如下(便于理解，仍然使用JDK1.7的环境)：

public class HashMapInfiniteLoop {

private static HashMap<Integer,String> map = new HashMap<Integer,String>(2，0.75f);

public static void main(String[] args) {

map.put(5， "C");

new Thread("Thread1") {

public void run() {

map.put(7, "B");

System.out.println(map);

};

}.start();

new Thread("Thread2") {

public void run() {

map.put(3, "A);

System.out.println(map);

};

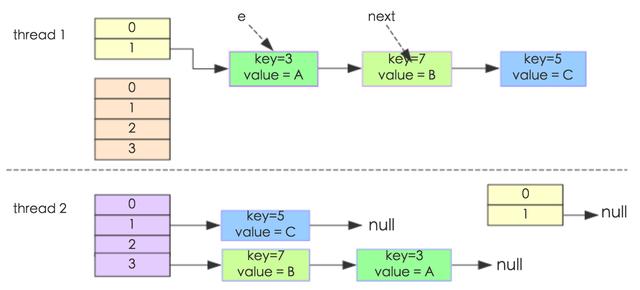
}.start();

}

}

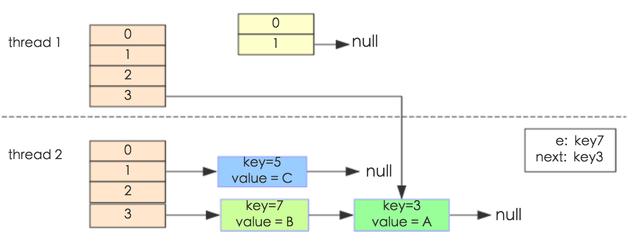
其中，map初始化为一个长度为2的数组，loadFactor=0.75，threshold=2\*0.75=1，也就是说当put第二个key的时候，map就需要进行resize。

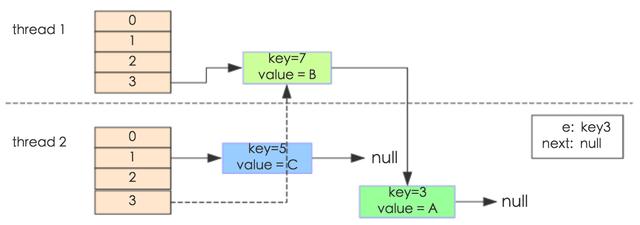
通过设置断点让线程1和线程2同时debug到transfer方法(3.3小节代码块)的首行。注意此时两个线程已经成功添加数据。放开thread1的断点至transfer方法的“Entry next = e.next;” 这一行；然后放开线程2的的断点，让线程2进行resize。结果如下图。



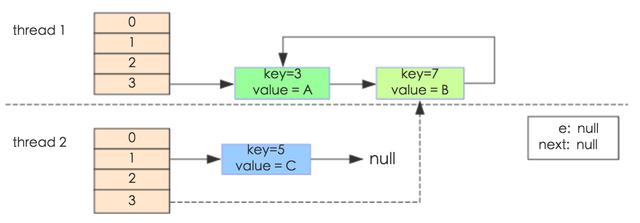
注意，Thread1的 e 指向了key(3)，而next指向了key(7)，其在线程二rehash后，指向了线程二重组后的链表。

线程一被调度回来执行，先是执行 newTalbe[i] = e， 然后是e = next，导致了e指向了key(7)，而下一次循环的next = e.next导致了next指向了key(3)。





e.next = newTable[i] 导致 key(3).next 指向了 key(7)。注意：此时的key(7).next 已经指向了key(3)， 环形链表就这样出现了。



于是，当我们用线程一调用map.get(11)时，悲剧就出现了——Infinite Loop。

### JDK1.8与JDK1.7的性能对比

HashMap中，如果key经过hash算法得出的数组索引位置全部不相同，即Hash算法非常好，那样的话，getKey方法的时间复杂度就是O(1)，如果Hash算法技术的结果碰撞非常多，假如Hash算极其差，所有的Hash算法结果得出的索引位置一样，那样所有的键值对都集中到一个桶中，或者在一个链表中，或者在一个红黑树中，时间复杂度分别为O(n)和O(lgn)。 鉴于JDK1.8做了多方面的优化，总体性能优于JDK1.7，下面我们从两个方面用例子证明这一点。

#### Hash较均匀的情况

为了便于测试，我们先写一个类Key，如下：

class Key implements Comparable<Key> {

private final int value;

Key(int value) {

this.value = value;

}

@Override

public int compareTo(Key o) {

return Integer.compare(this.value, o.value);

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass())

return false;

Key key = (Key) o;

return value == key.value;

}

@Override

public int hashCode() {

return value;

}

}

这个类复写了equals方法，并且提供了相当好的hashCode函数，任何一个值的hashCode都不会相同，因为直接使用value当做hashcode。为了避免频繁的GC，我将不变的Key实例缓存了起来，而不是一遍一遍的创建它们。代码如下：

public class Keys {

public static final int MAX\_KEY = 10\_000\_000;

private static final Key[] KEYS\_CACHE = new Key[MAX\_KEY];

static {

for (int i = 0; i < MAX\_KEY; ++i) {

KEYS\_CACHE[i] = new Key(i);

}

}

public static Key of(int value) {

return KEYS\_CACHE[value];

}

}

现在开始我们的试验，测试需要做的仅仅是，创建不同size的HashMap（1、10、100、……10000000），屏蔽了扩容的情况，代码如下：

static void test(int mapSize) {

HashMap<Key, Integer> map = new HashMap<Key,Integer>(mapSize);

for (int i = 0; i < mapSize; ++i) {

map.put(Keys.of(i), i);

}

long beginTime = System.nanoTime(); //获取纳秒

for (int i = 0; i < mapSize; i++) {

map.get(Keys.of(i));

}

long endTime = System.nanoTime();

System.out.println(endTime - beginTime);

}

public static void main(String[] args) {

for(int i=10;i<= 1000 0000;i\*= 10){

test(i);

}

}

在测试中会查找不同的值，然后度量花费的时间，为了计算getKey的平均时间，我们遍历所有的get方法，计算总的时间，除以key的数量，计算一个平均值，主要用来比较，绝对值可能会受很多环境因素的影响。结果如下：



通过观测测试结果可知，JDK1.8的性能要高于JDK1.7 15%以上，在某些size的区域上，甚至高于100%。由于Hash算法较均匀，JDK1.8引入的红黑树效果不明显，下面我们看看Hash不均匀的的情况。

#### Hash极不均匀的情况

假设我们又一个非常差的Key，它们所有的实例都返回相同的hashCode值。这是使用HashMap最坏的情况。代码修改如下：

class Key implements Comparable<Key> {

//...

@Override

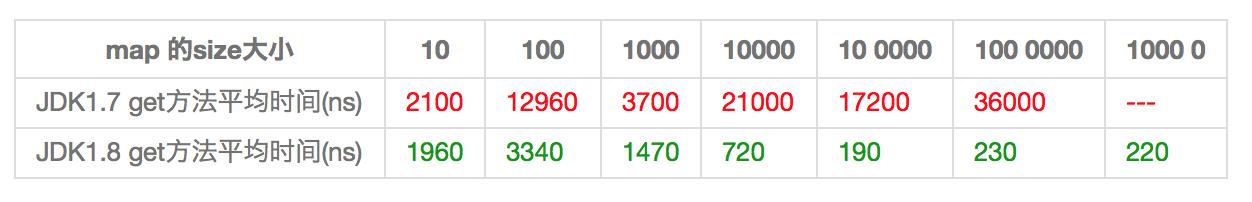
public int hashCode() {

return 1;

}

}

仍然执行main方法，得出的结果如下表所示：



从表中结果中可知，**随着size的变大，JDK1.7的花费时间是增长的趋势，而JDK1.8是明显的降低趋势，并且呈现对数增长稳定**。**当一个链表太长的时候，HashMap会动态的将它替换成一个红黑树**，这话的话会将时间复杂度从O(n)降为O(logn)。hash算法均匀和不均匀所花费的时间明显也不相同，这两种情况的相对比较，可以说明一个好的hash算法的重要性。

#### 小结

(1) 扩容是一个特别耗性能的操作，所以当程序员在使用HashMap的时候，估算map的大小，初始化的时候给一个大致的数值，避免map进行频繁的扩容。

(2) 负载因子是可以修改的，也可以大于1，但是建议不要轻易修改，除非情况非常特殊。

(3) HashMap是线程不安全的，不要在并发的环境中同时操作HashMap，建议使用ConcurrentHashMap。

(4) JDK1.8引入红黑树大程度优化了HashMap的性能。

(5) HashMap的性能提升仅仅是JDK1.8的冰山一角。

# 高并发编程系列

## ConcurrentHashMap的实现原理(JDK1.7和JDK1.8)

HashMap、CurrentHashMap 的实现原理基本都是BAT面试必考内容，[阿里P8架构师谈：深入探讨HashMap的底层结构、原理、扩容机制](http://youzhixueyuan.com/the-underlying-structure-and-principle-of-hashmap.html)深入谈过hashmap的实现原理以及在JDK 1.8的实现区别，今天主要谈CurrentHashMap的实现原理，以及在JDK1.7和1.8的区别。

**内容目录：**

1.哈希表

2.ConcurrentHashMap与HashMap、HashTable的区别

3.CurrentHashMap在JDK1.7和JDK1.8版本的区别

实现原理

### **哈希表**

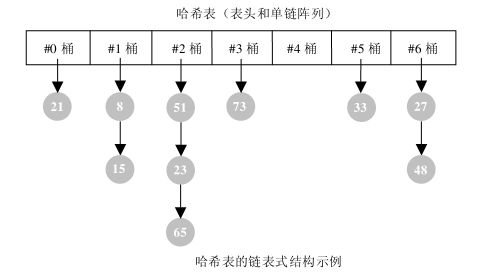
**1.介绍**

哈希表就是一种以 键-值(key-indexed) 存储数据的结构，我们只要输入待查找的值即key，即可查找到其对应的值。

哈希的思路很简单，如果所有的键都是整数，那么就可以使用一个简单的无序数组来实现：将键作为索引，值即为其对应的值，这样就可以快速访问任意键的值。这是对于简单的键的情况，我们将其扩展到可以处理更加复杂的类型的键。

**2.链式哈希表**

链式哈希表从根本上说是由一组链表构成。每个链表都可以看做是一个“桶”，我们将所有的元素通过散列的方式放到具体的不同的桶中。插入元素时，首先将其键传入一个哈希函数（该过程称为哈希键），函数通过散列的方式告知元素属于哪个“桶”，然后在相应的链表头插入元素。查找或删除元素时，用同们的方式先找到元素的“桶”，然后遍历相应的链表，直到发现我们想要的元素。因为每个“桶”都是一个链表，所以链式哈希表并不限制包含元素的个数。然而，如果表变得太大，它的性能将会降低。



**3.应用场景**

我们熟知的缓存技术（比如redis、memcached）的核心其实就是在内存中维护一张巨大的哈希表，还有大家熟知的HashMap、CurrentHashMap等的应用。

### **ConcurrentHashMap与HashMap等的区别**

**1.HashMap**

我们知道HashMap是线程不安全的，在多线程环境下，使用Hashmap进行put操作会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap。

**2.HashTable**

HashTable和HashMap的实现原理几乎一样，差别无非是

 HashTable不允许key和value为null

 HashTable是线程安全的

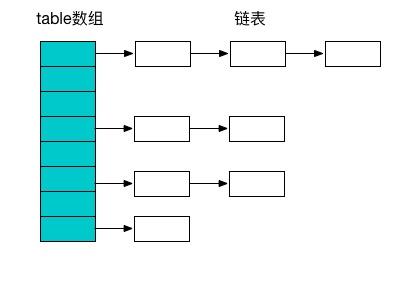
但是HashTable线程安全的策略实现代价却太大了，简单粗暴，get/put所有相关操作都是synchronized的，这相当于给整个哈希表加了一把大锁。

多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞，相当于将所有的操作串行化，在竞争激烈的并发场景中性能就会非常差。

**3.ConcurrentHashMap**

主要就是为了应对hashmap在并发环境下不安全而诞生的，ConcurrentHashMap的设计与实现非常精巧，大量的利用了volatile，final，CAS等lock-free技术来减少锁竞争对于性能的影响。

我们都知道Map一般都是数组+链表结构（JDK1.8该为数组+红黑树）。



ConcurrentHashMap避免了对全局加锁改成了局部加锁操作，这样就极大地提高了并发环境下的操作速度，由于ConcurrentHashMap在JDK1.7和1.8中的实现非常不同，接下来我们谈谈JDK在1.7和1.8中的区别。

### **JDK1.7版本的CurrentHashMap的实现原理**

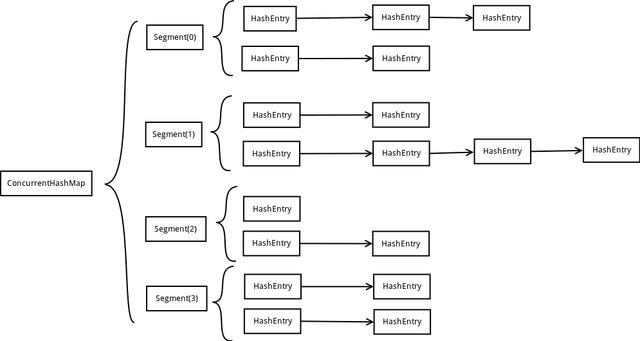
在JDK1.7中ConcurrentHashMap采用了数组+Segment+分段锁的方式实现。

**1.Segment(分段锁)**

ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表,同时又是一个ReentrantLock（Segment继承了ReentrantLock）。

**2.内部结构**

ConcurrentHashMap使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。如下图是ConcurrentHashMap的内部结构图：



从上面的结构我们可以了解到，ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作。

**第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部。**

**3.该结构的优劣势**

**坏处**

这一种结构的带来的副作用是Hash的过程要比普通的HashMap要长

**好处**

写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，这样，在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作（刚好这些写操作都非常平均地分布在所有的Segment上）。

所以，通过这一种结构，ConcurrentHashMap的并发能力可以大大的提高。

### JDK1.8版本的CurrentHashMap的实现原理

JDK8中ConcurrentHashMap参考了JDK8 HashMap的实现，采用了数组+链表+红黑树的实现方式来设计，内部大量采用CAS操作，这里我简要介绍下CAS。

CAS是compare and swap的缩写，即我们所说的比较交换。cas是一种基于锁的操作，而且是乐观锁。在java中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加version来获取数据，性能较悲观锁有很大的提高。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和A的值是一样的，那么就将内存里面的值更新成B。CAS是通过无限循环来获取数据的，若果在第一轮循环中，a线程获取地址里面的值被b线程修改了，那么a线程需要自旋，到下次循环才有可能机会执行。

**JDK8中彻底放弃了Segment转而采用的是Node，其设计思想也不再是JDK1.7中的分段锁思想。**

**Node：保存key，value及key的hash值的数据结构。其中value和next都用volatile修饰，保证并发的可见性。**

<strong>class Node<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

volatile V val;

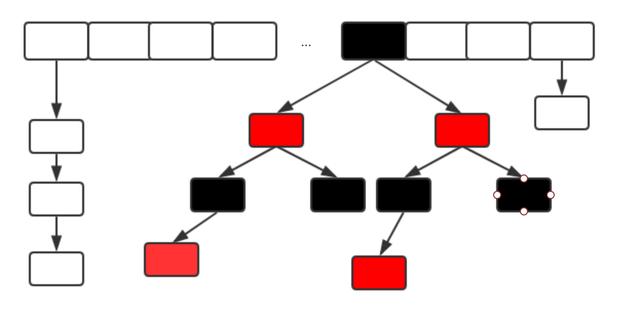
volatile Node<K,V> next;

//... 省略部分代码

} </strong>

**Java8 ConcurrentHashMap结构基本上和Java8的HashMap一样，不过保证线程安全性。**

在JDK8中ConcurrentHashMap的结构，由于引入了红黑树，使得ConcurrentHashMap的实现非常复杂，我们都知道，红黑树是一种性能非常好的二叉查找树，其查找性能为O（logN），但是其实现过程也非常复杂，而且可读性也非常差，Doug  
Lea的思维能力确实不是一般人能比的，早期完全采用链表结构时Map的查找时间复杂度为O（N），JDK8中ConcurrentHashMap在链表的长度大于某个阈值的时候会将链表转换成红黑树进一步提高其查找性能。



### 总结

其实可以看出JDK1.8版本的ConcurrentHashMap的数据结构已经接近HashMap，相对而言，ConcurrentHashMap只是增加了同步的操作来控制并发，从JDK1.7版本的ReentrantLock+Segment+HashEntry，到JDK1.8版本中synchronized+CAS+HashEntry+红黑树。

1.数据结构：取消了Segment分段锁的数据结构，取而代之的是数组+链表+红黑树的结构。  
2.保证线程安全机制：JDK1.7采用segment的分段锁机制实现线程安全，其中segment继承自ReentrantLock。JDK1.8采用CAS+Synchronized保证线程安全。  
3.锁的粒度：原来是对需要进行数据操作的Segment加锁，现调整为对每个数组元素加锁（Node）。  
4.链表转化为红黑树:定位结点的hash算法简化会带来弊端,Hash冲突加剧,因此在链表节点数量大于8时，会将链表转化为红黑树进行存储。  
5.查询时间复杂度：从原来的遍历链表O(n)，变成遍历红黑树O(logN)。

## NIO、BIO、AIO的区别，及NIO的应用和框架选型

### AIO、BIO、NIO的区别

**IO模型主要分类：**

* 同步(synchronous) IO和异步(asynchronous) IO
* 阻塞(blocking) IO和非阻塞(non-blocking)IO
* 同步阻塞(blocking-IO)简称BIO
* 同步非阻塞(non-blocking-IO)简称NIO
* 异步非阻塞(synchronous-non-blocking-IO)简称AIO

#### 1.BIO (同步阻塞I/O模式)

数据的读取写入必须阻塞在一个线程内等待其完成。

这里使用那个经典的烧开水例子，这里假设一个烧开水的场景，有一排水壶在烧开水，BIO的工作模式就是， 叫一个线程停留在一个水壶那，直到这个水壶烧开，才去处理下一个水壶。但是实际上线程在等待水壶烧开的时间段什么都没有做。

#### 2.NIO（同步非阻塞）

同时支持阻塞与非阻塞模式，但这里我们以其同步非阻塞I/O模式来说明，那么什么叫做同步非阻塞？如果还拿烧开水来说，NIO的做法是叫一个线程不断的轮询每个水壶的状态，看看是否有水壶的状态发生了改变，从而进行下一步的操作。

#### 3.AIO （异步非阻塞I/O模型）

异步非阻塞与同步非阻塞的区别在哪里？异步非阻塞无需一个线程去轮询所有IO操作的状态改变，在相应的状态改变后，系统会通知对应的线程来处理。对应到烧开水中就是，为每个水壶上面装了一个开关，水烧开之后，水壶会自动通知我水烧开了。

#### 4.IO与NIO区别：



#### 5.同步与异步的区别：

同步：发送一个请求，等待返回，再发送下一个请求，同步可以避免出现死锁，脏读的发生。

异步：发送一个请求，不等待返回，随时可以再发送下一个请求，可以提高效率，保证并发。

#### 6.阻塞和非阻塞

阻塞：传统的IO流都是阻塞式的。也就是说，当一个线程调用read()或者write()方法时，该线程将被阻塞，直到有一些数据读读取或者被写入，在此期间，该线程不能执行其他任何任务。在完成网络通信进行IO操作时，由于线程会阻塞，所以服务器端必须为每个客户端都提供一个独立的线程进行处理，当服务器端需要处理大量的客户端时，性能急剧下降。

非阻塞：Java  
NIO是非阻塞式的。当线程从某通道进行读写数据时，若没有数据可用时，该线程会去执行其他任务。线程通常将非阻塞IO的空闲时间用于在其他通道上执行IO操作，所以单独的线程可以管理多个输入和输出通道。因此NIO可以让服务器端使用一个或有限几个线程来同时处理连接到服务器端的所有客户端。

#### 7.BIO、NIO、AIO适用场景

* BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择。
* NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂。
* AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

### NIO的3个核心概念

NIO重点是把**Channel（通道），Buffer（缓冲区），Selector（选择器）**三个类之间的关系弄清楚。

#### 1.缓冲区Buffer

Buffer是一个对象。它包含一些要写入或者读出的数据。在面向流的I/O中，可以将数据写入或者将数据直接读到Stream对象中。

**在NIO中，所有的数据都是用缓冲区处理。**这也就本文上面谈到的**IO是面向流的，NIO是面向缓冲区的。**

缓冲区实质是一个数组，通常它是一个字节数组（ByteBuffer），也可以使用其他类的数组。但是一个缓冲区不仅仅是一个数组，缓冲区提供了对数据的结构化访问以及维护读写位置（limit）等信息。

最常用的缓冲区是ByteBuffer，一个ByteBuffer提供了一组功能于操作byte数组。除了ByteBuffer，还有其他的一些缓冲区，事实上，每一种Java基本类型（除了Boolean）都对应一种缓冲区，具体如下：

* ByteBuffer：字节缓冲区
* CharBuffer:字符缓冲区
* ShortBuffer：短整型缓冲区
* IntBuffer：整型缓冲区
* LongBuffer:长整型缓冲区
* FloatBuffer：浮点型缓冲区
* DoubleBuffer：双精度浮点型缓冲区

#### 2.通道Channel

Channel是一个通道，可以通过它读取和写入数据，他就像自来水管一样，网络数据通过Channel读取和写入。

通道和流不同之处在于通道是双向的，流只是在一个方向移动，而且通道可以用于读，写或者同时用于读写。

因为Channel是全双工的，所以它比流更好地映射底层操作系统的API，特别是在UNIX网络编程中，底层操作系统的通道都是全双工的，同时支持读和写。

**Channel有四种实现：**

* FileChannel:是从文件中读取数据。
* DatagramChannel:从UDP网络中读取或者写入数据。
* SocketChannel:从TCP网络中读取或者写入数据。
* ServerSocketChannel:允许你监听来自TCP的连接，就像服务器一样。每一个连接都会有一个SocketChannel产生。

#### 3.多路复用器Selector

Selector选择器可以监听多个Channel通道感兴趣的事情(read、write、accept(服务端接收)、connect，实现一个线程管理多个Channel，节省线程切换上下文的资源消耗。Selector只能管理非阻塞的通道，FileChannel是阻塞的，无法管理。

**关键对象**

* Selector：选择器对象，通道注册、通道监听对象和Selector相关。
* SelectorKey：通道监听关键字，通过它来监听通道状态。

**监听注册**

监听注册在Selector

socketChannel.register(selector, SelectionKey.OP\_READ);

**监听的事件有**

* OP\_ACCEPT: 接收就绪，serviceSocketChannel使用的
* OP\_READ: 读取就绪，socketChannel使用
* OP\_WRITE: 写入就绪，socketChannel使用
* OP\_CONNECT: 连接就绪，socketChannel使用

### NIO的应用和框架



#### NIO的应用

Java NIO成功的应用在了各种分布式、即时通信和中间件Java系统中，充分的证明了基于NIO构建的通信基础，是一种高效，且扩展性很强的通信架构。

例如：Dubbo(服务框架)，就默认使用Netty作为基础通信组件，用于实现各进程节点之间的内部通信。

Jetty、Mina、Netty、Dubbo、ZooKeeper等都是基于NIO方式实现。

* Mina出身于开源界的大牛Apache组织
* Netty出身于商业开源大亨Jboss
* Dubbo阿里分布式服务框架

#### NIO框架

特别是Netty是目前最流行的一个Java开源框架NIO框架，Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。

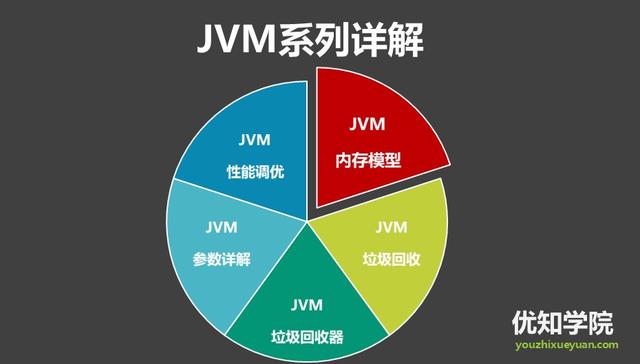
相比JDK原生NIO，Netty提供了相对十分简单易用的API，非常适合网络编程。

Mina和Netty这两个NIO框架的创作者是同一个人Trustin Lee 。Netty从某种程度上讲是Mina的延伸和扩展，解决了一些Mina上的设计缺陷，也优化了一下Mina上面的设计理念。

**另一方面Netty相比较Mina的优势：**

1. 更容易学习
2. API更简单
3. 详细的范例源码和API文档
4. 更活跃的论坛和社区
5. 更高的代码更新维护速度

**Netty无疑是NIO框架的首选**，它的健壮性、功能、性能、可定制性和可扩展性在同类框架都是首屈一指的，后续将重点详细谈Netty的实现原理以及实战场景。



JVM基本是BAT面试必考的内容，今天我们先从JVM内存模型开启详解整个JVM系列，希望看完整个系列后，可以轻松通过BAT关于JVM的考核。

## 4大并发工具类详解CountDownLatch、Semaphore等

通常我们所说的并发包也就是java.util.concurrent，Java并发编程集中在这个并发包种，主要包含：Java并发工具类和并发容器等。

今天主要介绍concurrent包的内容以及4大并发工具类。

### 并发工具包



#### **并发工具类**

提供了比synchronized更加高级的各种同步结构：包括CountDownLatch、CyclicBarrier、Semaphore等，可以实现更加丰富的多线程操作。

#### **并发容器**

提供各种线程安全的容器：最常见的ConcurrentHashMap、有序的ConcurrentSkipListMap,实现线程安全的动态数组CopyOnWriteArrayList等。

#### **并发队列**

各种BlockingQueue的实现：常用的ArrayBlockingQueue、SynchorousQueue或针对特定场景的PriorityBlockingQueue。

#### **Executor框架**

可以创建各种不同类型的线程池，调度任务运行等，绝大部分情况下，不再需要自己从头实现线程池和任务调度器。

### 常用的并发容器



**1.ConcurrentHashMap**

经常使用的并发容器，JDK 1.7和1.8的底层数据结构发生了变化，这里可以建议学习顺序如下：从Java7 HashMap -> Java7 ConcurrentHashMap -> Java8 HashMap -> Java8 ConcurrentHashMap，这样可以更好的掌握这个并发容器，毕竟都是从HashMap进化而来。

**2.ConcurrentSkipListMap**

在乎顺序，需要对数据进行非常频繁的修改

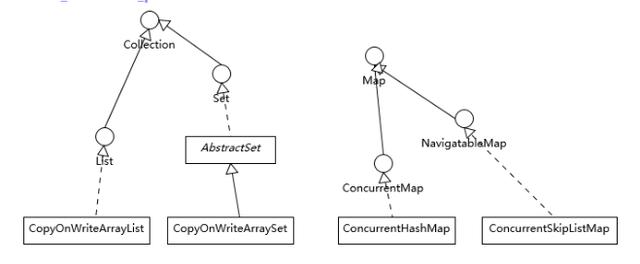
**3.CopyOnWrite容器**

CopyOnWrite容器即写时复制的容器。从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器,CopyOnWriteArrayList和CopyOnWriteArraySet。

**4.各种并发队列的实现**

如各种BlockedQueue实现，比较典型的ArrayBlockingQueue、SynchorousQueue。

详情请看：[高并发编程系列：并发容器的原理，7大并发容器详解、及使用场景](http://youzhixueyuan.com/use-of-concurrent-containers.html)



### 常用的并发工具类



**1.CountDownLatch**

**功能**

CountDownLatch是一个同步的辅助类，允许一个或多个线程，等待其他一组线程完成操作，再继续执行。

原理：

* CountDownLatch是通过一个计数器来实现的，计数器的初始值为需要等待线程的数量。

eg：CountDownLatch c = new CountDownLatch(10); // 等待线程的数量为10

* 主线程调用CountDownLatch的await()方法会阻塞当前线程(即:主线程在闭锁上等待)，直到计数器的值为0。
* 当一个工作线程完成了自己的任务后，调用CountDownLatch的countDown()方法，计数器的值就会减1。
* 当计数器值为0时，说明所有的工作线程都执行完了，此时，在闭锁上等待的主线程就可以恢复执行任务。

**应用场景**

倒数计时器

例如：一种典型的场景就是火箭发射。在火箭发射前，为了保证万无一失，往往还要进行各项设备、仪器的检查。 只有等所有检查完毕后，引擎才能点火。这种场景就非常适合使用CountDownLatch。

它可以使得点火线程，等待所有检查线程全部完工后，再执行

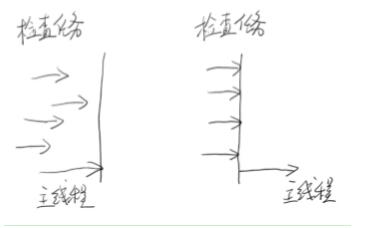
**使用方式**

static final CountDownLatch end = new CountDownLatch(10);

end.countDown();

end.await();

**示意图：**



**2.CyclicBarrier**

**功能:**

CyclicBarrier的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。

和CountDownLatch相似，也是等待某些线程都做完以后再执行。

**与CountDownLatch区别**

在于这个计数器可以反复使用。比如，假设我们将计数器设置为10。那么凑齐第一批1 0个线程后，计数器就会归零，然后接着凑齐下一批10个线程。

**原理：**

1)CyclicBarrier是通过一个计数器来实现的，计数器的初始值为需要等待线程的数量。eg：CyclicBarrier c = new CyclicBarrier(2); // 等待线程的数量为2

2)每个线程调用CyclicBarrier的await()方法，使自己进入等待状态。

3)当所有的线程都调用了CyclicBarrier的await()方法后，所有的线程停止等待，继续运行。

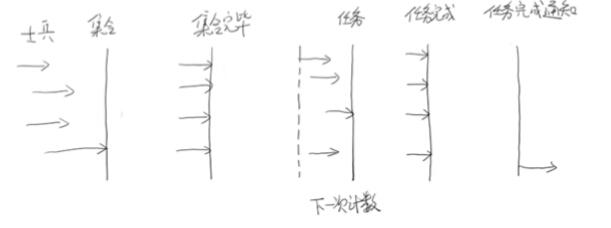
**使用方式：**

public CyclicBarrier(int parties, Runnable barrierAction)

barrierAction就是当计数器一次计数完成后，系统会执行的动作

await()

**示意图：**



**3.信号量Semaphore**

功能：Java提供了经典信号量Semaphore的实现，它通过控制一定数量的许可（permit）的方式，来达到限制通用资源访问的目的。例如：控制并发的线程数。

**原理：**

1)Semaphore是通过一个计数器(记录许可证的数量)来实现的，计数器的初始值为需要等待线程的数量。

eg：Semaphore s = new Semaphore(10); // 线程最大的并发数为10

2)线程通过acquire()方法获取许可证(计数器的值减1)，只有获取到许可证才可以继续执行下去，否则阻塞当前线程。

3)线程通过release()方法归还许可证(计数器的值加1)。

说明：使用tryAcquire()方法可以立即得到执行的结果：尝试获取一个许可证，若获取成功，则立即返回true，若获取失败，则立即返回false。

**应用场景：**

Semaphore可以用于做流量控制，特别是公用资源有限的应用场景，比如数据库连接。

举一个场景：例如在车站、机场等出租车时，当很多空出租车就位时，为防止过度拥挤，调度员指挥排队等待坐车的队伍一次进来5个人上车，等这5个人坐车出发，再放进去下一批。这和Semaphore的工作原理有些类似。

**4.交换者Exchanger**

功能：Exchanger（交换者）是一个用于线程间协作的工具类。Exchanger用于进行线程间的数据交换。它提供一个同步点，在这个同步点两个线程可以交换彼此的数据。这两个线程通过exchange方法交换数据，  
如果第一个线程先执行exchange方法，它会一直等待第二个线程也执行exchange，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。

原理：

1)线程A调用public V exchange(V dataA)方法，线程A到达同步点，并且在线程B到达同步点前一直等待。

2)线程B调用public V exchange(V dataB)方法，线程B到达同步点。

3)线程A与线程B都达到同步点时，线程将自己的数据传递给对方，两个线程完成了数据的交换了。

**Exchanger的应用场景**

Exchanger可以用于校对工作的场景。

## 4种常用Java线程锁的特点，性能比较、使用场景

### **多线程的缘由**

在出现了进程之后，操作系统的性能得到了大大的提升。虽然进程的出现解决了操作系统的并发问题，但是人们仍然不满足，人们逐渐对实时性有了要求。

使用多线程的理由之一是和进程相比，它是一种非常花销小，切换快，更”节俭”的多任务操作方式。

在Linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种”昂贵”的多任务工作方式。而在进程中的同时运行多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程间彼此切换所需的时间也远远小于进程间切换所需要的时间。

### 多线程并发面临的问题



由于多个线程是共同占有所属进程的资源和地址空间的，那么就会存在一个问题：

如果多个线程要同时访问某个资源，怎么处理？

在Java并发编程中，经常遇到多个线程访问同一个 共享资源 ，这时候作为开发者必须考虑如何维护数据一致性，这就是Java锁机制(同步问题)的来源。

**Java提供了多种多线程锁机制的实现方式，常见的有：**

1. synchronized
2. ReentrantLock
3. Semaphore
4. AtomicInteger等

每种机制都有优缺点与各自的适用场景，必须熟练掌握他们的特点才能在Java多线程应用开发时得心应手。

### 4种Java线程锁(线程同步)



**1.synchronized**

在Java中synchronized关键字被常用于维护数据一致性。

synchronized机制是给共享资源上锁，只有拿到锁的线程才可以访问共享资源，这样就可以强制使得对共享资源的访问都是顺序的。

Java开发人员都认识synchronized，使用它来实现多线程的同步操作是非常简单的，只要在需要同步的对方的方法、类或代码块中加入该关键字，它能够保证在同一个时刻最多只有一个线程执行同一个对象的同步代码，可保证修饰的代码在执行过程中不会被其他线程干扰。使用synchronized修饰的代码具有原子性和可见性，在需要进程同步的程序中使用的频率非常高，可以满足一般的进程同步要求。

synchronized (obj) {

//方法

…….

}

synchronized实现的机理依赖于软件层面上的JVM，因此其性能会随着Java版本的不断升级而提高。

到了Java1.6，synchronized进行了很多的优化，有适应自旋、锁消除、锁粗化、轻量级锁及偏向锁等，效率有了本质上的提高。在之后推出的Java1.7与1.8中，均对该关键字的实现机理做了优化。

需要说明的是，当线程通过synchronized等待锁时是不能被Thread.interrupt()中断的，因此程序设计时必须检查确保合理，否则可能会造成线程死锁的尴尬境地。

最后，尽管Java实现的锁机制有很多种，并且有些锁机制性能也比synchronized高，但还是强烈推荐在多线程应用程序中使用该关键字，因为实现方便，后续工作由JVM来完成，可靠性高。只有在确定锁机制是当前多线程程序的性能瓶颈时，才考虑使用其他机制，如ReentrantLock等。

**2.ReentrantLock**

可重入锁，顾名思义，这个锁可以被线程多次重复进入进行获取操作。

ReentantLock继承接口Lock并实现了接口中定义的方法，除了能完成synchronized所能完成的所有工作外，还提供了诸如可响应中断锁、可轮询锁请求、定时锁等避免多线程死锁的方法。

Lock实现的机理依赖于特殊的CPU指定，可以认为不受JVM的约束，并可以通过其他语言平台来完成底层的实现。在并发量较小的多线程应用程序中，ReentrantLock与synchronized性能相差无几，但在高并发量的条件下，synchronized性能会迅速下降几十倍，而ReentrantLock的性能却能依然维持一个水准。

**因此我们建议在高并发量情况下使用ReentrantLock。**

ReentrantLock引入两个概念：公平锁与非公平锁。

公平锁指的是锁的分配机制是公平的，通常先对锁提出获取请求的线程会先被分配到锁。反之，JVM按随机、就近原则分配锁的机制则称为不公平锁。

ReentrantLock在构造函数中提供了是否公平锁的初始化方式，默认为非公平锁。这是因为，非公平锁实际执行的效率要远远超出公平锁，除非程序有特殊需要，否则最常用非公平锁的分配机制。

ReentrantLock通过方法lock()与unlock()来进行加锁与解锁操作，与synchronized会被JVM自动解锁机制不同，ReentrantLock加锁后需要手动进行解锁。为了避免程序出现异常而无法正常解锁的情况，使用ReentrantLock必须在finally控制块中进行解锁操作。通常使用方式如下所示：

Lock lock = new ReentrantLock();

try {

lock.lock();

//…进行任务操作5 }

finally {

lock.unlock();

}

**3.Semaphore**

上述两种锁机制类型都是“互斥锁”，学过操作系统的都知道，互斥是进程同步关系的一种特殊情况，相当于只存在一个临界资源，因此同时最多只能给一个线程提供服务。但是，在实际复杂的多线程应用程序中，可能存在多个临界资源，这时候我们可以借助Semaphore信号量来完成多个临界资源的访问。

Semaphore基本能完成ReentrantLock的所有工作，使用方法也与之类似，通过acquire()与release()方法来获得和释放临界资源。

经实测，Semaphone.acquire()方法默认为可响应中断锁，与ReentrantLock.lockInterruptibly()作用效果一致，也就是说在等待临界资源的过程中可以被Thread.interrupt()方法中断。

此外，Semaphore也实现了可轮询的锁请求与定时锁的功能，除了方法名tryAcquire与tryLock不同，其使用方法与ReentrantLock几乎一致。Semaphore也提供了公平与非公平锁的机制，也可在构造函数中进行设定。

Semaphore的锁释放操作也由手动进行，因此与ReentrantLock一样，为避免线程因抛出异常而无法正常释放锁的情况发生，释放锁的操作也必须在finally代码块中完成。

**4.AtomicInteger**

首先说明，此处AtomicInteger是一系列相同类的代表之一，常见的还有AtomicLong、AtomicLong等，他们的实现原理相同，区别在与运算对象类型的不同。

我们知道，在多线程程序中，诸如++i  
或  
i++等运算不具有原子性，是不安全的线程操作之一。通常我们会使用synchronized将该操作变成一个原子操作，但JVM为此类操作特意提供了一些同步类，使得使用更方便，且使程序运行效率变得更高。通过相关资料显示，通常AtomicInteger的性能是ReentantLock的好几倍。

### Java线程锁总结

**1.synchronized：**

在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized是很合适的。原因在于，编译程序通常会尽可能的进行优化synchronize，另外可读性非常好。

**2.ReentrantLock:**

在资源竞争不激烈的情形下，性能稍微比synchronized差点点。但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍，而ReentrantLock确还能维持常态。

**高并发量情况下使用ReentrantLock。**

**3.Atomic:**

和上面的类似，不激烈情况下，性能比synchronized略逊，而激烈的时候，也能维持常态。激烈的时候，Atomic的性能会优于ReentrantLock一倍左右。但是其有一个缺点，就是只能同步一个值，一段代码中只能出现一个Atomic的变量，多于一个同步无效。因为他不能在多个Atomic之间同步。

所以，我们写同步的时候，优先考虑synchronized，如果有特殊需要，再进一步优化。ReentrantLock和Atomic如果用的不好，不仅不能提高性能，还可能带来灾难。

以上就是Java线程锁的详解，除了从编程的角度应对高并发，更多还需要从架构设计的层面来应对高并发场景，例如：Redis缓存、CDN、异步消息等，详细的内容如下。



## Java锁详解：独享锁/共享锁+公平锁/非公平锁+乐观锁/悲观锁

在Java并发场景中，会涉及到各种各样的锁如公平锁，乐观锁，悲观锁等等，这篇文章介绍各种锁的分类：

公平锁/非公平锁

可重入锁

独享锁/共享锁

乐观锁/悲观锁

分段锁

自旋锁

### 乐观锁 VS 悲观锁

乐观锁与悲观锁是一种广义上的概念，体现了看待线程同步的不同角度，在Java和数据库中都有此概念对应的实际应用。

#### **乐观锁**

顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。

**乐观锁适用于多读的应用类型**，乐观锁在Java中是通过使用无锁编程来实现，**最常采用的是CAS算法**，Java原子类中的递增操作就通过CAS自旋实现的。

CAS全称 Compare And Swap（比较与交换），是一种无锁算法。在不使用锁（没有线程被阻塞）的情况下实现多线程之间的变量同步。java.util.concurrent包中的原子类就是通过CAS来实现了乐观锁。

简单来说，CAS算法有3个三个操作数：

* 需要读写的内存值 V。
* 进行比较的值 A。
* 要写入的新值 B。

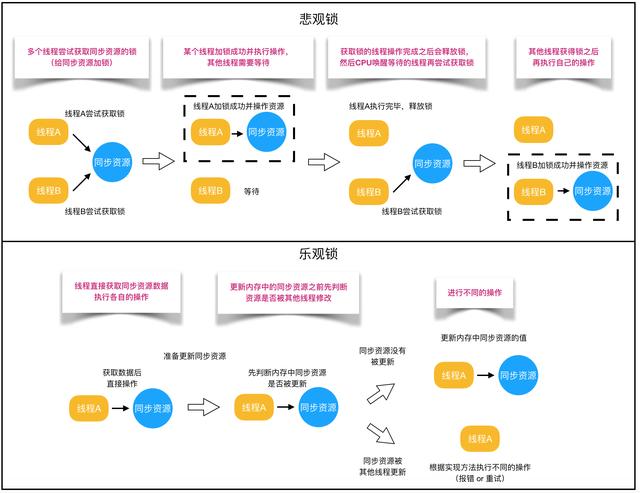
**当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则返回V**。这是一种乐观锁的思路，它相信在它修改之前，没有其它线程去修改它；而**Synchronized是一种悲观锁，它认为在它修改之前，一定会有其它线程去修改它，悲观锁效率很低**。

#### **悲观锁**

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。

传统的MySQL关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。详情可以参考：[阿里P8架构师谈：MySQL行锁、表锁、悲观锁、乐观锁的特点与应用](http://youzhixueyuan.com/mysql-row-locks-table-locks-pessimistic-locks.html)

再比如上面提到的Java的同步synchronized关键字的实现就是典型的悲观锁。



#### **3.总之：**

* 悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确。
* 乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升。

### 公平锁 VS 非公平锁

#### **公平锁**

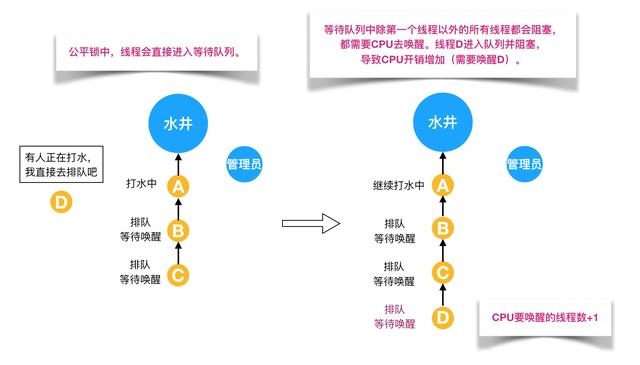
就是很公平，在并发环境中，每个线程在获取锁时会先查看此锁维护的等待队列，如果为空，或者当前线程是等待队列的第一个，就占有锁，否则就会加入到等待队列中，以后会按照FIFO的规则从队列中取到自己。

**公平锁的优点**是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU唤醒阻塞线程的开销比非公平锁大。

#### **非公平锁**

上来就直接尝试占有锁，如果尝试失败，就再采用类似公平锁那种方式。

非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU不必唤醒所有线程。缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁。



**3.典型应用:**

java jdk并发包中的ReentrantLock可以指定构造函数的boolean类型来创建公平锁和**非公平锁（默认）**,比如：公平锁可以使用new ReentrantLock(true)实现。

### 独享锁 VS 共享锁

**1.独享锁**

是指该锁一次只能被一个线程所持有。

**2.共享锁**

是指该锁可被多个线程所持有。

**3.比较**

对于Java ReentrantLock而言，其是独享锁。但是对于Lock的另一个实现类ReadWriteLock，其读锁是共享锁，其写锁是独享锁。

**读锁的共享锁可保证并发读是非常高效的**，读写，写读 ，写写的过程是互斥的。

**独享锁与共享锁也是通过AQS来实现的**，通过实现不同的方法，来实现独享或者共享。

**4.AQS**

抽象队列同步器（Abstract Queued Synchronizer，简称AQS）是用来构建锁或者其他同步组件的基础框架，它使用一个整型的volatile变量（命名为state）来维护同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作。



concurrent包的实现结构如上图所示，AQS、非阻塞数据结构和原子变量类等基础类都是基于volatile变量的读/写和CAS实现，而像Lock、同步器、阻塞队列、Executor和并发容器等高层类又是基于基础类实现。

### **分段锁**

分段锁其实是一种锁的设计，并不是具体的一种锁，对于ConcurrentHashMap而言，其并发的实现就是通过分段锁的形式来实现高效的并发操作。

我们以ConcurrentHashMap来说一下分段锁的含义以及设计思想，ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap（JDK7与JDK8中HashMap的实现）的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock)。

当需要put元素的时候，并不是对整个hashmap进行加锁，而是先通过hashcode来知道他要放在那一个分段中，然后对这个分段进行加锁，所以当多线程put的时候，只要不是放在一个分段中，就实现了真正的并行的插入。

但是，在统计size的时候，可就是获取hashmap全局信息的时候，就需要获取所有的分段锁才能统计。

**分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。**

**Java线程锁**

详细可以参考：[高并发编程系列：4种常用Java线程锁的特点，性能比较、使用场景](http://youzhixueyuan.com/4-kinds-of-java-thread-locks.html)

## 4大JVM性能分析工具详解，及内存泄漏分析方案

### **谈到性能优化分析一般会涉及到：**

Java代码层面的，典型的循环嵌套等

还会涉及到Java JVM：内存泄漏溢出等

MySQL数据库优化：分库分表、慢查询、长事务的优化等

[阿里P8架构师谈：MySQL慢查询优化、索引优化、以及表等优化总结](http://youzhixueyuan.com/mysql-slow-query-optimization-index-optimization.html)

[阿里P8架构师谈：架构设计之数据库垂直、水平拆分六大原则](http://youzhixueyuan.com/six-principles-of-vertical-and-horizontal-resolution-of-database.html)(数据库分库分表)

[常用的后端性能优化六种方式：缓存化+服务化+异步化等](http://youzhixueyuan.com/six-ways-to-optimize-rear-end-performance.html)

[阿里P8架构师谈：MySQL数据库的索引原理、与慢SQL优化的5大原则](http://youzhixueyuan.com/index-principle-and-slow-query-optimization-of-mysql.html)

[阿里P8架构师谈：Web前端、应用服务器、数据库SQL等性能优化总结](http://youzhixueyuan.com/web-front-end-application-database-performance-optimization.html)

[阿里P8架构师谈：多线程、架构、异步消息、Redis等性能优化策略](http://youzhixueyuan.com/high-concurrency-performance-optimization-method.html)

今天主要分享JVM性能调优工具，文末有详细的JVM调优方法和步骤。

### **什么场景需要JVM调优**

1. OutOfMemoryError，内存不足
2. 内存泄露
3. 线程死锁
4. 锁争用（Lock Contention）
5. Java进程消耗CPU过高

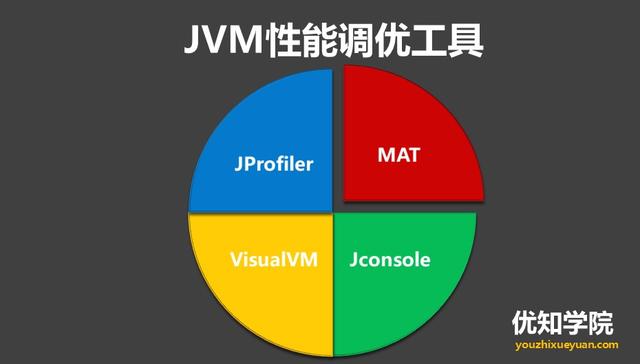
这些问题出现的时候常常通过重启服务器或者调大内存来临时解决，实际情况，还需要尽量还原当时的业务场景，并分析内存、线程等数据，通过分析找到最终的解决方案，这就会涉及到性能分析工具。

### **JVM性能监控分析工具**

JDK本身提供了很丰富的性能监控工具，除了集成式的visualVM和jConsole外，还有jstat,jstack,jps,jmap,jhat小工具，这些都是性能调优的常用工具。

1. Jconsole : jdk自带，功能简单，但是可以在系统有一定负荷的情况下使用。对垃圾回收算法有很详细的跟踪。
2. JProfiler：商业软件，功能强大。
3. VisualVM：JDK自带，功能强大，与JProfiler类似。
4. MAT：MAT(Memory Analyzer Tool)，一个基于Eclipse的内存分析工具。

**下面介绍这些JVM工具的使用**。



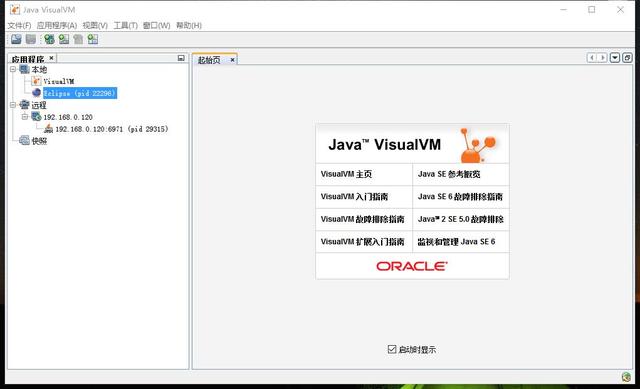
JVM性能调优工具

### **VisualVM**

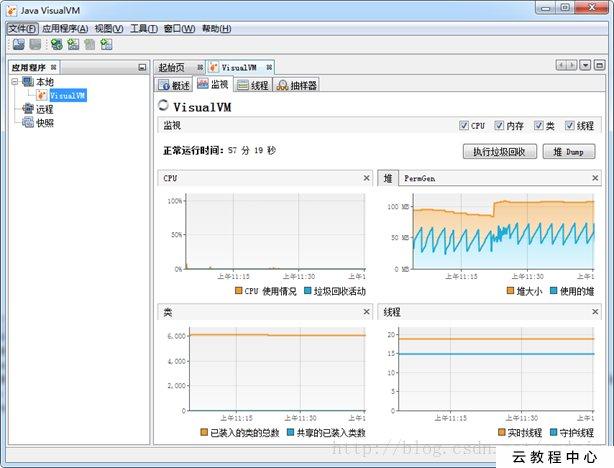
VisualVM 是javajdk自带的牛逼的调优工具，也是平时使用最多调优工具，几乎涉及了jvm调优的方方面面。启动起来后和jconsole 一样同样可以选择本地和远程，如果需要监控远程同样需要配置相关参数。

**1 打开VisualVM**

这个工具放在JDK安装目录的bin目录下，双击jvisualvm.exe即可打开，如下图所示



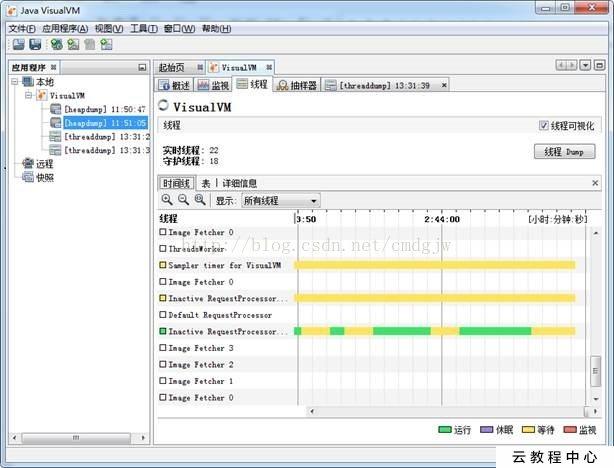
**2. 监视页面主要展示 系统资源占用情况**



* CPU :展示java程序运行的时候占用的cpu资源
* 堆 :这里要说明下堆内存的组成部分,堆是由老年代和新生代组成，其中新生代有由”伊甸园”和”两个幸存区组成”三部分组成，堆视图看到的资源占用实际是”老年代”、”伊甸园（Eden）”、”两个幸存者（Survivor ）”的一个综合情况。
* PermGen :Perm 区用来存放java类以及其他虚拟机自己的静态数据，（常被称为持久代或者方法区）
* 类 :此视图 主要展示 当前程序加载了多少个类
* 线程: 当前程序的线程启动情况
* 堆Dump : 生产当前程序的内存快照hprof文件，对于分析内存溢出问题比较有帮助。

**3. 线程页面(主要展示程序中所有的线程运行状态)**

* 线程dump ： 所有线程的快照（对分析线程死锁，比较有帮助）
* 时间线 ： 展示每个线程的实时运行状态（不同颜色代表不同的状态）



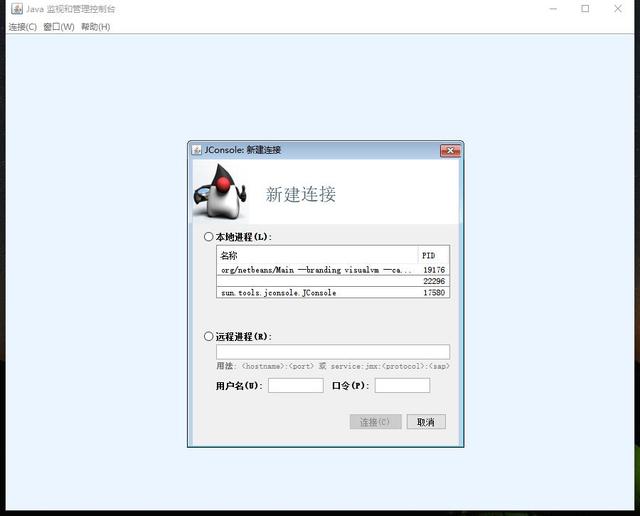
**VisualVM可以根据需要安装不同的插件，每个插件的关注点都不同，有的主要监控GC，有的主要监控内存，有的监控线程等。**

### **Jconsole**

JConsole是一个JMX（Java  
Management  
Extensions，即Java管理扩展）的JVM监控与管理工具，监控主要体现在：堆栈内存、线程、CPU、类、VM信息这几个方面，而管理主要是对JMX  
MBean（managed  
beans，被管理的beans，是一系列资源，包含对象、接口、设备等）的管理，不仅能查看bean的属性和方法信息，还能够在运行时修改属性或调用方法。

**1.打开Jconsole**

直接在jdk/bin目录下点击jconsole.exe即可启动，界面如下:



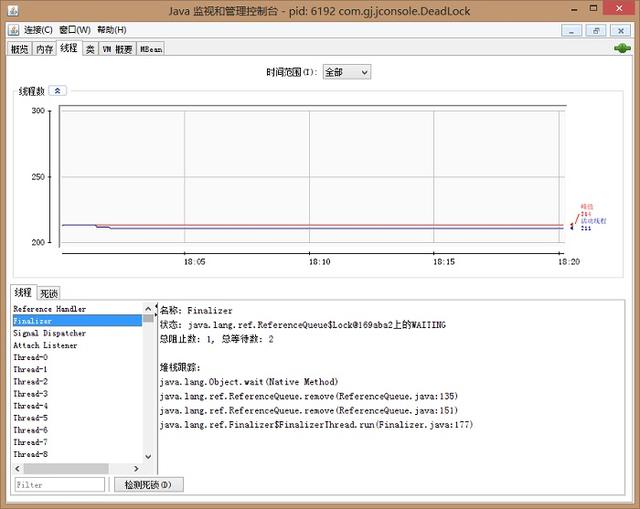
**1、内存**

在内存页我们可以看到程序运行期间JVM各个部分的内存状况，右下角是对应各个分区的内存使用柱状图，点击对应柱可查看详情，看图：



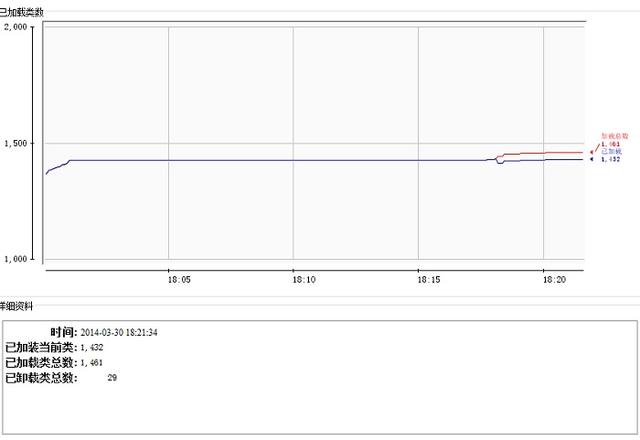
**2、线程**

该页面可以查看当前JVM进程启动了多少个线程，并能查看每个线程的状态及堆栈信息，此外还有一个功能就是能够自动检测死锁，见图：



**3、类**

该页面其实和线程页有些相似，不过显示的是JVM加载类的信息，见图：

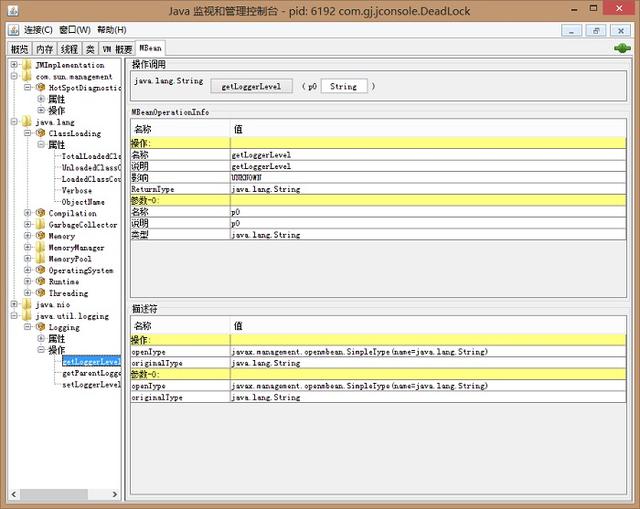


**4、VM概述**

这个其实没必要细说，看图就明白，显示了当前JVM的各方面信息：



**5、MBean管理**

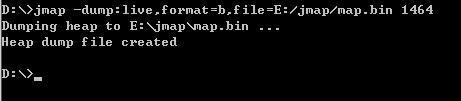


### **MAT**

MAT(Memory Analyzer Tool)，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗。

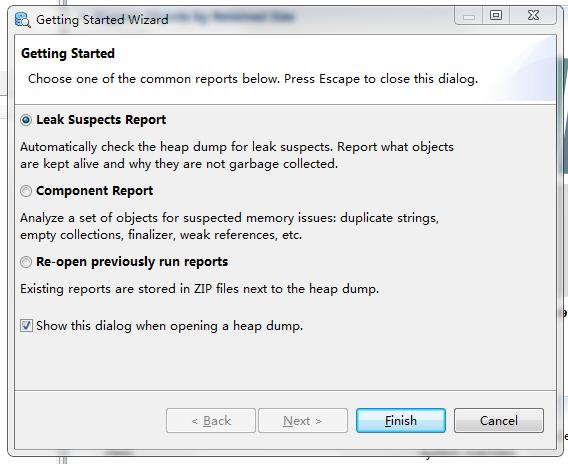
可以利用visualvm或者是 jmap命令生产堆文件在进行内存分析。

**1. 用jmap生成堆信息**



这样在E盘的jmap文件夹里会有一个map.bin的堆信息文件

**2. 将堆信息导入到mat中分析**

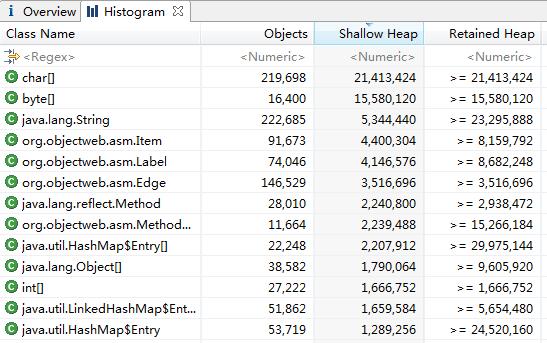


**3. 生成分析报告**

可以利用visualvm或者是 jmap命令生产堆文件，导入eclipse mat中生成分析报告：

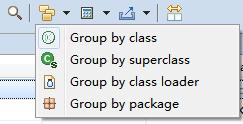


**1)Histogram（直方图）视图**



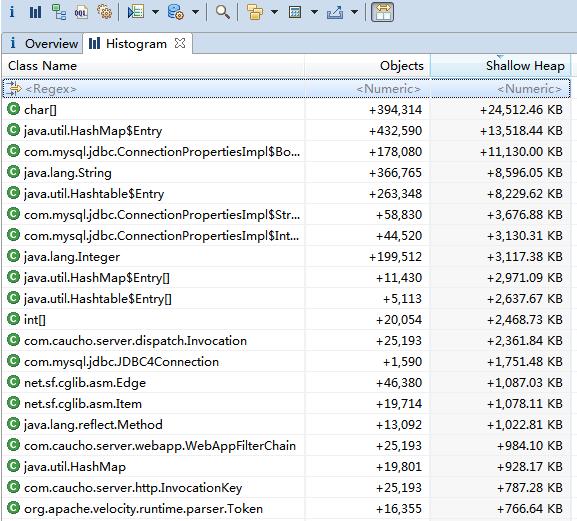
* Class Name ： 类名称，java类名
* Objects ： 类的对象的数量，这个对象被创建了多少个
* Shallow Heap ：一个对象内存的消耗大小，不包含对其他对象的引用
* Retained Heap ：是shallow Heap的总和，也就是该对象被GC之后所能回收到内存的总和

通过直方图视图可以很容易找到占用内存最多的几个类（通过Retained Heap排序），还可以通过其他方式进行分组（见下图）。



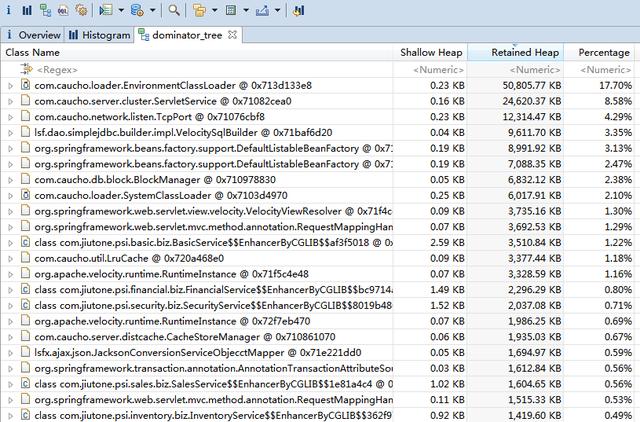
如果存在内存溢出，时间久了溢出类的实例数量或者内存占比会越来越多，排名也越来越靠前。

图标进行对比，通过多次对比不同时间点下的直方图对比就很容易把溢出的类找出来。

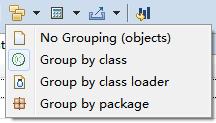


**2)支配树（Dominator Tree）**

MAT提供了一个称为支配树（Dominator Tree）的对象图。支配树体现了对象实例间的支配关系，在此视图中列出了每个对象（Object Instance）与其引用关系的树状结构，同时包含了占用内存的大小和百分比。



**通过Dominator Tree视图可以很容易的找出占用内存最多的几个对象**（根据Retained Heap或Percentage排序），和Histogram类似，可以通过不同的方式进行分组显示：



Histogram视图和Dominator Tree视图的角度不同，前者是基于类的角度，后者是基于对象实例的角度，并且可以更方便的看出其引用关系。

以上只是一个初步的介绍，mat还有更强大的使用，比如对比堆内存，在生产环境中往往为了定位问题，每隔几分钟dump出一下内存快照，随后在对比不同时间的堆内存的变化来发现问题。

### **JVM内存泄漏分析**

性能调优方法：

1. 生成dump文件
2. mat性能Fenix
3. 分析内存泄露
4. 代码及参数调整

造成OutOfMemoryError内存泄露典型原因：对象已经死了，无法通过垃圾收集器进行自动回收，需要通过找出泄露的代码位置和原因，才好确定解决方案。

**分析步骤：**

1. 用工具生成java应用程序的heap dump（如jmap）

2. 使用Java heap分析工具（如MAT），找出内存占用超出预期的嫌疑对象

3. 根据情况，分析嫌疑对象和其他对象的引用关系。

4. 分析程序的源代码，找出嫌疑对象数量过多的原因。

**JVM调优分析和参数设置请参考**：[JVM性能调优的6大步骤，及关键调优参数详解](http://youzhixueyuan.com/jvm-performance-optimization.html)

## 并发容器的原理，7大并发容器详解、及使用场景

### 并发容器的由来

在Java并发编程中，经常听到Java集合类，同步容器、并发容器，那么他们有哪些具体分类，以及各自之间的区别和优劣呢？

**只有把这些梳理清楚了，你才能真正掌握在高并发的环境下，正确使用好并发容器，我们先从Java集合类，同步容器谈起。**



**1.什么是同步容器**

Java的集合容器框架中，主要有四大类别：List、Set、Queue、Map，大家熟知的这些集合类ArrayList、LinkedList、HashMap这些容器都是非线程安全的。

如果有多个线程并发地访问这些容器时，就会出现问题。因此，在编写程序时，在多线程环境下必须要求程序员手动地在任何访问到这些容器的地方进行同步处理，这样导致在使用这些容器的时候非常地不方便。

所以，Java先提供了同步容器供用户使用。

**同步容器可以简单地理解为通过synchronized来实现同步的容器**，比如Vector、Hashtable以及SynchronizedList等容器。

**2.同步容器，主要的分类**：

* Vector
* Stack
* HashTable
* Collections.synchronized方法生成

### 同步容器面临的问题

可以通过查看Vector，Hashtable等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，**并在需要同步的方法上加上关键字synchronized。**

这样做的代价是削弱了并发性，当多个线程共同竞争容器级的锁时，吞吐量就会降低。

例如： HashTable只要有一条线程获取了容器的锁之后，其他所有的线程访问同步函数都会被阻塞，因此同一时刻只能有一条线程访问同步函数。

**因此为了解决同步容器的性能问题，所以才有了并发容器。**

### 什么是并发容器

java.util.concurrent包中提供了多种并发类容器。

并发类容器是专门针对多线程并发设计的，使用了**锁分段技术**，只对操作的位置进行同步操作，但是其他没有操作的位置其他线程仍然可以访问，提高了程序的吞吐量。

**采用了CAS算法和部分代码使用synchronized锁保证线程安全。**

### 并发容器有哪些分类



**1.ConcurrentHashMap**

对应的非并发容器：HashMap

目标：代替Hashtable、synchronizedMap，支持复合操作

原理：JDK6中采用一种更加细粒度的加锁机制Segment“分段锁”，JDK8中采用CAS无锁算法。

**2.CopyOnWriteArrayList**

对应的非并发容器：ArrayList

目标：代替Vector、synchronizedList

原理：利用高并发往往是读多写少的特性，对读操作不加锁，对写操作，先复制一份新的集合，在新的集合上面修改，然后将新集合赋值给旧的引用，并通过volatile 保证其可见性，当然写操作的锁是必不可少的了。

**3.CopyOnWriteArraySet**

对应的非并发容器：HashSet

目标：代替synchronizedSet

原理：基于CopyOnWriteArrayList实现，其唯一的不同是在add时调用的是CopyOnWriteArrayList的addIfAbsent方法，其遍历当前Object数组，如Object数组中已有了当前元素，则直接返回，如果没有则放入Object数组的尾部，并返回。

**4.ConcurrentSkipListMap**

对应的非并发容器：TreeMap

目标：代替synchronizedSortedMap(TreeMap)

原理：Skip list（跳表）是一种可以代替平衡树的数据结构，默认是按照Key值升序的。

**5.ConcurrentSkipListSet**

对应的非并发容器：TreeSet

目标：代替synchronizedSortedSet

原理：内部基于ConcurrentSkipListMap实现

**6.ConcurrentLinkedQueue**

不会阻塞的队列

对应的非并发容器：Queue

原理：基于链表实现的FIFO队列（LinkedList的并发版本）

**7.LinkedBlockingQueue、ArrayBlockingQueue、PriorityBlockingQueue**

对应的非并发容器：BlockingQueue

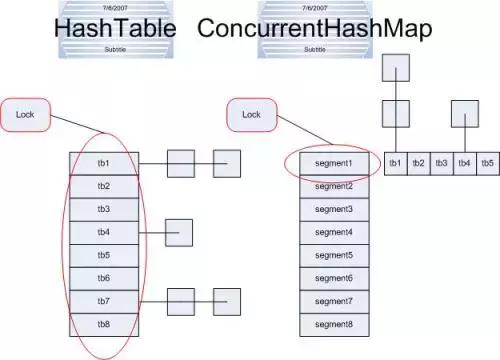
特点：拓展了Queue，增加了可阻塞的插入和获取等操作

原理：通过ReentrantLock实现线程安全，通过Condition实现阻塞和唤醒

实现类：

* LinkedBlockingQueue：基于链表实现的可阻塞的FIFO队列
* ArrayBlockingQueue：基于数组实现的可阻塞的FIFO队列
* PriorityBlockingQueue：按优先级排序的队列

### **ConcurrentHashMap的实现**



HashMap,Hashtable与ConcurrentHashMap都是实现的哈希表数据结构，在随机读取的时候效率很高。

Hashtable实现同步是利用synchronized关键字进行锁定的，其是针对整张哈希表进行锁定的，即每次锁住整张表让线程独占，在线程安全的背后是巨大的浪费。

ConcurrentHashMap和Hashtable主要区别就是围绕着锁的粒度进行区别以及如何区锁定。

上图中，左边是Hashtable的实现方式，可以看到锁住整个哈希表；而右边则是ConcurrentHashMap的实现方式，单独锁住每一个桶（segment）.ConcurrentHashMap将哈希表分为16个桶（默认值），诸如get(),put(),remove()等常用操作只锁当前需要用到的桶,而size()才锁定整张表。

原来只能一个线程进入，现在却能同时接受16个写线程并发进入（写线程需要锁定，而读线程几乎不受限制）。

**所以，才有了并发性的极大提升。**

高并发编程，除了并发容器，还会涉及到并发工具类:CountDownLatch等，后续将详细的介绍并发工具类，以及ConcurrentHashMap的底层实现细节，不仅要知其然,还要知其所以然，这样才能更好的掌握好高并发编程。

## Java线程池的使用方式，核心运行原理、以及注意事项

[高并发编程系列：4种常用Java线程锁的特点，性能比较、使用场景](http://youzhixueyuan.com/4-kinds-of-java-thread-locks.html)

### 线程池的缘由

java中为了提高并发度，可以使用多线程共同执行,但是如果有大量线程短时间之内被创建和销毁，会占用大量的系统时间，影响系统效率。

为了解决上面的问题，**java中引入了线程池**，可以使创建好的线程在指定的时间内由系统统一管理，而不是在执行时创建，执行后就销毁，从而避免了频繁创建、销毁线程带来的系统开销。

**线程池如何使用，以及实现原理，处理步骤，有什么使用注意事项等，今天主要从这几个方面详细介绍Java线程池。**



### 线程池的处理流程

就以ThreadPoolExecutor为例，当我们把一个Runnable交给线程池去执行的时候，这个线程池处理的流程是这样的：



1. 先判断线程池中的核心线程们是否空闲，如果空闲，就把这个新的任务指派给某一个空闲线程去执行。如果没有空闲，并且当前线程池中的核心线程数还小于 corePoolSize，那就再创建一个核心线程。
2. 如果线程池的线程数已经达到核心线程数，并且这些线程都繁忙，就把这个新来的任务放到等待队列中去。如果等待队列又满了，那么查看一下当前线程数是否到达maximumPoolSize，如果还未到达，就继续创建线程。
3. 如果已经到达了，就交给RejectedExecutionHandler(拒绝策略)来决定怎么处理这个任务。

### 线程池的使用(ThreadPoolExecutor)

在Java中，线程池的概念是Executor这个接口，具体实现为ThreadPoolExecutor类，是线程池中最核心的一个类，因此如果要透彻地了解Java中的线程池，必须先了解这个类。

**ThreadPoolExecutor继承了AbstractExecutorService类，并提供了四个构造器**：

public class ThreadPoolExecutor extends AbstractExecutorService {

…..

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler);

…

}

ThreadPoolExecutor继承了AbstractExecutorService类，并提供了四个构造器，事实上，通过观察每个构造器的源码具体实现，发现前面三个构造器都是调用的第四个构造器进行的初始化工作。

**下面解释下一下构造器中各个参数的含义：**

**1.corePoolSize（线程池的基本大小）**

当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有基本线程。

**2.runnableTaskQueue（任务队列）**

用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。

* ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。
* LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。
* SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQueue。
* PriorityBlockingQueue：一个具有优先级得无限阻塞队列。

**3.maximumPoolSize（线程池最大大小）**

线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。

**4.ThreadFactory：用于设置创建线程的工厂**

可以通过线程工厂给每个创建出来的线程设置更有意义的名字，Debug和定位问题时非常又帮助。

**5.RejectedExecutionHandler（饱和策略）**

当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。以下是JDK1.5提供的四种策略。n AbortPolicy：直接抛出异常。

* CallerRunsPolicy：只用调用者所在线程来运行任务。
* DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
* DiscardPolicy：不处理，丢弃掉。
* 当然也可以根据应用场景需要来实现RejectedExecutionHandler接口自定义策略。如记录日志或持久化不能处理的任务。

**6.keepAliveTime（线程活动保持时间）**

线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。

**7.TimeUnit（线程活动保持时间的单位）**

可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)，微秒(MICROSECONDS, 千分之一毫秒)和毫微秒(NANOSECONDS, 千分之一微秒)。

### 线程池的注意事项

虽然线程池能大大提高服务器的并发性能，但使用它也会存在一定风险。与所有多线程应用程序一样，用线程池构建的应用程序容易产生各种并发问题，如对共享资源的竞争和死锁。此外，如果线程池本身的实现不健壮，或者没有合理地使用线程池，还容易导致与线程池有关的死锁、系统资源不足和线程泄漏等问题。

**1) 建议使用new ThreadPoolExecutor(…)的方式创建线程池**

线程池的创建不应使用  
Executors 去创建，而应该通过 ThreadPoolExecutor  
创建，这样可以让读者更加明确地知道线程池的参数设置、运行规则，规避资源耗尽的风险，这一点在也阿里巴巴JAVA开发手册中也有明确要求。这一点不容小觑，曾有同学因为线程池使用不当导致生产的同一台机器上部署的多个应用都因无法创建线程池而出现故障。

**2) 合理设置线程数**

线程池的工作线程数设置应根据实际情况配置，CPU密集型业务（搜索、排序等）CPU空闲时间较少，线程数不能设置太多。

如果是CPU密集型任务，就需要尽量压榨CPU，参考值可以设为 NCPU+1

如果是IO密集型任务，参考值可以设置为2\*NCPU

**3) 设置能代表具体业务的线程名称**

这样方便通过日志的线程名称识别所属业务。具体实现可以通过指定ThreadPoolExecutor的ThreadFactory参数。如使Spring提供的CustomizableThreadFactory。

以上就是Java线程池的详细介绍，除了从编程的角度应对高并发，更多还需要**从架构设计**的层面来应对高并发场景，例如：Redis缓存、MySQL数据库的优化、异步消息等



# 高并发架构系列

1、分布式锁+时间戳

2、消息队列

## Redis并发竞争key的解决方案详解

### Redis高并发的问题

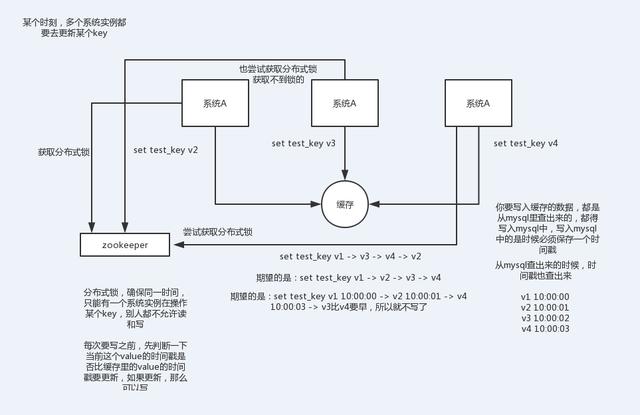
Redis缓存的高性能有目共睹，应用的场景也是非常广泛，但是在高并发的场景下，也会出现问题：

高并发架构系列：Redis缓存和MySQL数据一致性方案详解

如何解决Redis缓存雪崩、缓存穿透、缓存并发等5大难题

以及今天要谈到的Redis并发竞争问题，这里的并发指的是多个redis的client同时set key引起的并发问题。

比如：多客户端同时并发写一个key，一个key的值是1，本来按顺序修改为2,3,4，最后是4，但是由于并发设置的原因，最后顺序变成了4,3,2，最后变成的key值成了2。



### 如何解决Redis的并发竞争key问题

#### **第一种方案：分布式锁**

**1.整体技术方案**

这种情况，主要是准备一个分布式锁，大家去抢锁，抢到锁就做set操作。

**2.为什么是分布式锁**

因为传统的加锁的做法（如java的synchronized和Lock）这里没用，只适合单点。因为这是分布式环境，需要的是分布式锁。

当然，分布式锁可以基于很多种方式实现，比如zookeeper、redis等，不管哪种方式实现，基本原理是不变的：用一个状态值表示锁，对锁的占用和释放通过状态值来标识。

**3.分布式锁的要求**

* 互斥性：在任意一个时刻，只有一个客户端持有锁。
* 无死锁：即便持有锁的客户端崩溃或者其他意外事件，锁仍然可以被获取。
* 容错：只要大部分Redis节点都活着，客户端就可以获取和释放锁

**4.分布式锁的实现方式**

* 数据库
* Memcached（add命令）
* Redis（setnx命令）
* Zookeeper（临时节点）

具体的分布式锁实现，请参考：[阿里P8架构师谈：分布式锁的3种实现（数据库、缓存、Zookeeper)](http://youzhixueyuan.com/3-implementations-of-distributed-locks.html)

#### **第二种方案：利用消息队列**

在并发量过大的情况下,可以通过消息中间件进行处理,把并行读写进行串行化。

把Redis.set操作放在队列中使其串行化,必须的一个一个执行。

这种方式在一些高并发的场景中算是一种通用的解决方案。

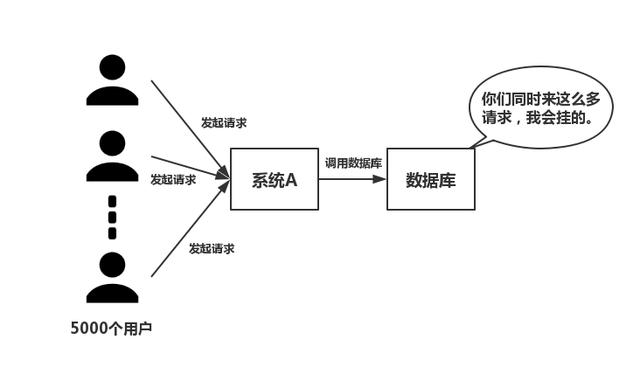
## **Redis缓存和MySQL数据一致性方案详解**

redis数据一致性解决方案

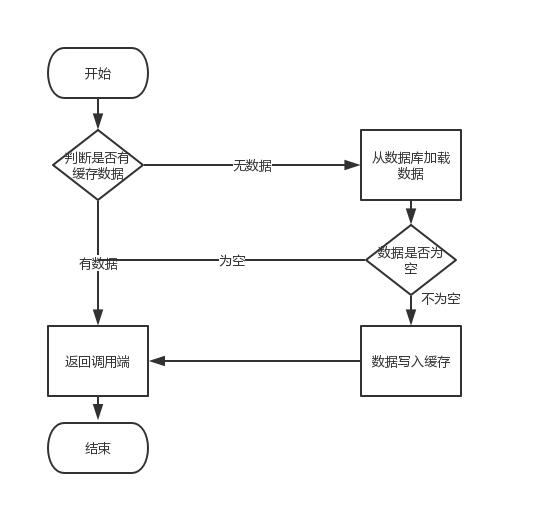
延时双删策略、binlog异步更新缓存

### 需求起因

在高并发的业务场景下，数据库大多数情况都是用户并发访问最薄弱的环节。所以，就需要使用redis做一个缓冲操作，让请求先访问到redis，而不是直接访问MySQL等数据库。



这个业务场景，主要是解决读数据从Redis缓存，一般都是按照下图的流程来进行业务操作。



读取缓存步骤一般没有什么问题，但是一旦涉及到数据更新：数据库和缓存更新，就容易出现**缓存(Redis)和数据库（MySQL）间的数据一致性问题**。

不管是先写MySQL数据库，再删除Redis缓存；还是先删除缓存，再写库，都有可能出现数据不一致的情况。举一个例子：

1.如果删除了缓存Redis，还没有来得及写库MySQL，另一个线程就来读取，发现缓存为空，则去数据库中读取数据写入缓存，此时缓存中为脏数据。

2.如果先写了库，在删除缓存前，写库的线程宕机了，没有删除掉缓存，则也会出现数据不一致情况。

因为写和读是并发的，没法保证顺序,就会出现缓存和数据库的数据不一致的问题。

如来解决？这里给出两个解决方案，先易后难，结合业务和技术代价选择使用。

**缓存和数据库一致性解决方案**

### **第一种方案：采用延时双删策略**

在写库前后都进行redis.del(key)操作，并且设定合理的超时时间。

伪代码如下

public void write(String key,Object data){

redis.delKey(key);

db.updateData(data);

Thread.sleep(500);

redis.delKey(key);

}

**2.具体的步骤就是：**

1）先删除缓存

2）再写数据库

3）休眠500毫秒

4）再次删除缓存

**那么，这个500毫秒怎么确定的，具体该休眠多久呢？**

需要评估自己的项目的读数据业务逻辑的耗时。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

当然这种策略还要考虑redis和数据库主从同步的耗时。最后的的写数据的休眠时间：则在读数据业务逻辑的耗时基础上，加几百ms即可。比如：休眠1秒。

**3.设置缓存过期时间**

从理论上来说，给缓存设置过期时间，是保证最终一致性的解决方案。所有的写操作以数据库为准，只要到达缓存过期时间，则后面的读请求自然会从数据库中读取新值然后回填缓存。

**4.该方案的弊端**

结合双删策略+缓存超时设置，这样最差的情况就是在超时时间内数据存在不一致，而且又增加了写请求的耗时。

### **第二种方案：异步更新缓存(基于订阅binlog的同步机制)**

**1.技术整体思路：**

MySQL binlog增量订阅消费+消息队列+增量数据更新到redis

**1）读Redis**：热数据基本都在Redis

**2）写MySQL**:增删改都是操作MySQL

**3）更新Redis数据**：MySQ的数据操作binlog，来更新到Redis

**2.Redis更新**

**1）数据操作主要分为两大块：**

* 一个是全量(将全部数据一次写入到redis)
* 一个是增量（实时更新）

这里说的是增量,指的是mysql的update、insert、delate变更数据。

**2）读取binlog后分析 ，利用消息队列,推送更新各台的redis缓存数据。**

这样一旦MySQL中产生了新的写入、更新、删除等操作，就可以把binlog相关的消息推送至Redis，Redis再根据binlog中的记录，对Redis进行更新。

其实这种机制，很类似MySQL的主从备份机制，因为MySQL的主备也是通过binlog来实现的数据一致性。

这里可以结合使用canal(阿里的一款开源框架)，通过该框架可以对MySQL的binlog进行订阅，而canal正是模仿了mysql的slave数据库的备份请求，使得Redis的数据更新达到了相同的效果。

当然，这里的消息推送工具你也可以采用别的第三方：kafka、rabbitMQ等来实现推送更新Redis。

以上就是Redis和MySQL数据一致性详解，相关的MySQL数据库主从同步一致性可以参考：[高并发架构系列：数据库主从同步的3种一致性方案实现，及优劣比较](http://youzhixueyuan.com/database-master-slave-synchronization.html)

## **解决Redis缓存雪崩、缓存穿透、缓存并发等5大难题**

Java相关的面试都会问到缓存的问题：[史上最全Redis面试49题(含答案):哨兵+复制+事务+集群+持久化等](http://youzhixueyuan.com/redis-interview-question-49-answers.html)，除此之外还会问到缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级等不常见的问题，但却是非常重要的问题，今天谈谈这个话题。

### 缓存解决方案：

缓存预热、缓存雪崩、缓存降级、缓存穿透。

### **缓存雪崩**

数据未加载到缓存中，或者缓存同一时间大面积的失效，从而导致所有请求都去查数据库，导致数据库CPU和内存负载过高，甚至宕机。

**比如一个雪崩的简单过程：**

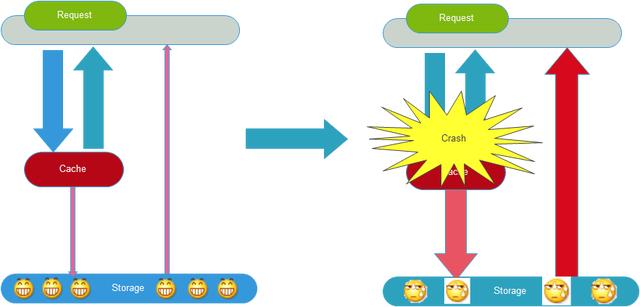
1、redis集群大面积故障

2、缓存失效，但依然大量请求访问缓存服务redis

3、redis大量失效后，大量请求转向到mysql数据库

4、mysql的调用量暴增，很快就扛不住了，甚至直接宕机

5、由于大量的应用服务依赖mysql和redis的服务，这个时候很快会演变成各服务器集群的雪崩，最后网站彻底崩溃。



#### **如何预防缓存雪崩：**

高可用、提前演练、服务降级、缓存备份、快速预热。

**1.缓存的高可用性**

缓存层设计成高可用，防止缓存大面积故障。即使个别节点、个别机器、甚至是机房宕掉，依然可以提供服务，例如 Redis Sentinel 和 Redis Cluster 都实现了高可用。

**2.缓存降级**

可以利用ehcache等本地缓存(暂时支持)，但主要还是对源服务访问进行限流、资源隔离（熔断）、降级等。

当访问量剧增、服务出现问题仍然需要保证服务还是可用的。系统可以根据一些关键数据进行**自动降级**，也可以配置开关实现**人工降级，**这里会涉及到运维的配合。

**降级的最终目的是保证核心服务可用，即使是有损的。**

比如推荐服务中，很多都是个性化的需求，假如个性化需求不能提供服务了，可以降级补充热点数据，不至于造成前端页面是个大空白。

在进行降级之前要对系统进行梳理，比如：哪些业务是核心(必须保证)，哪些业务可以容许暂时不提供服务(利用静态页面替换)等，以及配合服务器核心指标，来后设置整体预案，比如：

（1）一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

（2）警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；

（3）错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

（4）严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

**3.Redis备份和快速预热**

1)Redis数据备份和恢复

2)快速缓存预热

**4.提前演练**

最后，建议还是在项目上线前，演练缓存层宕掉后，应用以及后端的负载情况以及可能出现的问题，对高可用提前预演，提前发现问题。

### **缓存穿透**

缓存穿透是指查询一个一不存在的数据。例如：从缓存redis没有命中，需要从mysql数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

解决思路：

如果查询数据库也为空，直接设置一个默认值存放到缓存，这样第二次到缓冲中获取就有值了，而不会继续访问数据库。设置一个过期时间或者当有值的时候将缓存中的值替换掉即可。

可以给key设置一些格式规则，然后查询之前先过滤掉不符合规则的Key。

### **缓存并发**

这里的并发指的是多个redis的client同时set  
key引起的并发问题。其实redis自身就是单线程操作，多个client并发操作，按照先到先执行的原则，先到的先执行，其余的阻塞。当然，另外的解决方案是把redis.set操作放在队列中使其串行化，必须的一个一个执行。

### **缓存预热**

缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。

这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

解决思路：

1、直接写个缓存刷新页面，上线时手工操作下；

2、数据量不大，可以在项目启动的时候自动进行加载；

目的就是在系统上线前，将数据加载到缓存中。

以上就是缓存雪崩、预热、降级等的介绍，更多整体从服务器雪崩的角度，具体可以参考：[阿里P8架构师谈：什么是缓存雪崩？服务器雪崩的场景与解决方案](http://youzhixueyuan.com/scenarios-and-solutions-for-cache-and-server-avalanche.html)

## **大型网站Web前端优化最佳实践，以及最全优化工具集锦**

前端的性能对于一个Web应用来说非常重要，如果一个Web应用的页面加载速度非常快、对于用户的操作可以及时响应，那么产品的用户体验将会极大地提升。下图显示了页面加载速度对于用户体验的影响。

### 大型网站Web优化最佳实践



**1. 避免坏请求**

有时页面中的HTML或CSS会向服务器请求一个不存在的资源，比如图片或HTML文件，这会造成浏览器与服务器之间过多的往返请求，类似于：

浏览器：“我需要这个图像。”

服务器：“我没有这个图像。”

浏览器：“你确定吗？这个文档说你有。”

服务器：“真的没有。”

如此一来，会降低页面的加载速度。因此，检查页面中的坏链接非常有必要，你可以通过 Google的PageSpeed工具 来检测，找到问题后，补充相应的资源文件或者修改资源的链接地址即可。

**2. 避免CSS @import**

使用 @import方法引用CSS文件可以能会带来一些影响页面加载速度的问题，比如导致文件按顺序加载（一个加载完后才会加载另一个），而无法并行加载。

你可以使用 CSS delivery工具 来检测页面代码中是否存在@import方法。比如，如果检测结果中存在

**[css]** view plaincopy

1. @import url(“style.css”)

则建议你使用下面的代码来替代。

**[html]** view plaincopy

1. **<link** rel=”style.css” href=”style.css” type=”text/css”**>**

**3. 避免使用document.write**

在JavaScript中，可以使用 document.write在网页上显示内容或调用外部资源，而通过此方法，浏览器必须采取一些多余的步骤——下载资源、读取资源、运行JavaScript来了解需要做什么，调用其他资源时需要重新再执行一次这个过程。由于浏览器之前不知道要显示什么，所以会降低页面加载的速度。

要知道，任何能够被document.write调用的资源，都可以通过HTML来调用，这样速度会更快。检查你的页面代码，如果存在类似于下面的代码：

**[js]** view plaincopy

1. document.write(‘<script src=”another.js”></script>’);

建议修改为：

**[html]** view plaincopy

1. **<script** src=”another.js”**></script>**

**4. 合并多个外部CSS文件**

在网站中每使用一个CSS文件，都会让你的页面加载速度慢一点点。如果你有一个以上的CSS文件，你应该将它们合并为一个文件。

你可以通过 CSS delivery工具 来检测页面代码中的CSS文件，然后通过复制粘贴的方式将它们合并为一个。合并后记得修改页面中的引用代码，并删除旧的引用代码。

**5. 合并多个外部JavaScript文件**

大部分情况下，网站往往会包含若干个 JavaScript文件，但并不需要将这些文件都独立出来，其中有些是可以合并为一个文件的。

你可以通过 resource check工具 来检测页面中所引用的JavaScript文件数，然后可以通过复制粘贴的方式将多个文件合并为一个。

**6. 通过CSS sprites来整合图像**

如果页面中有6个小图像，那么浏览器在显示时会分别下载。你可以通过CSS sprites将这些图像合并成1个，可以减少页面加载所需的时间。

CSS sprites需要有两个步骤：整合图像、定位图像。比如你可以通过下面的代码来分别定位下面图像中的上下两部分。

**[css]** view plaincopy

1. .megaphone {**width**:50px; **height**:50px; **background**:url(images/sprite.png) 0 0px;}
2. .smile {**width**:50px; **height**:50px; **background**:url(images/sprite.png) 0 -50px;}

**7. 延迟JavaScript的加载**

浏览器在执行JavaScript代码时会停止处理页面，当页面中有很多JavaScript文件或代码要加载时，将导致严重的延迟。尽管可以使用defer、异步或将JavaScript代码放到页面底部来延迟JavaScript的加载，但这些都不是一个好的解决方案。

下面是Google的建议。

**[js]** view plaincopy

1. <script type=”text/javascript”>
2. **function** downloadJSAtOnload() {
3. **var** element = document.createElement(“script”);
4. element.src = “defer.js”;
5. document.body.appendChild(element);
6. }
7. **if** (window.addEventListener)
8. window.addEventListener(“load”, downloadJSAtOnload, **false**);
9. **else** **if** (window.attachEvent)
10. window.attachEvent(“onload”, downloadJSAtOnload);
11. **else** window.onload = downloadJSAtOnload;
12. </script>

这段代码的意思是等待页面加载完成后，然后再加载外部的“defer.js”文件。下面是测试结果。

**8. 启用压缩/ GZIP**

使用gzip对HTML和CSS文件进行压缩，通常可以节省大约50％到70％的大小，这样加载页面只需要更少的带宽和更少的时间。

你可以通过这个 Gzip压缩工具 来检测页面是否已经经过Gzip压缩。

**9. 启用Keep-Alive**

HTTP协议采用“请求-应答”模式，当使用普通模式（非KeepAlive模式）时，每个请求/应答客户和服务器都要新建一个连接，完成之后立即断开连接（HTTP协议为无连接的协议）；当使用 Keep-Alive模式（又称持久连接、连接重用）时，Keep-Alive功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive功能避免了建立或者重新建立连接。

在HTTP 1.0中Keep-Alive默认是关闭的，需要在HTTP头中加入“Connection: Keep-Alive”，才能启用Keep-Alive；在 HTTP1.1中Keep-Alive默认启用，加入“Connection: close”可关闭。目前大部分浏览器都是用HTTP 1.1协议，也就是说默认都会发起Keep-Alive的连接请求了，所以是否能完成一个完整的Keep- Alive连接就看Web服务器的设置情况。

**10. 将小的CSS和JavaScript代码内嵌到HTML中**

如果你的CSS代码比较小，可以将这部分代码放到HTML文件中，而不是一个外部CSS文件，这样可以减少页面加载所需的文件数，从而加快页面的加载。同样，也可以将小的 JavaScript脚本代码内嵌到HTML文件中。

**[html]** view plaincopy

1. **<style** type=”text/css”**>**
2. <!–CSS代码–>
3. **</style>**
4. **<script** type=”text/javascript”**>**
5. <!–JavaScript代码–>
6. **</script>**

**11. 利用浏览器缓存**

在显示页面时，浏览器需要加载logo、CSS文件和其他一些资源。浏览器缓存所做的工作就是“记住”已经加载的资源，让页面的加载速度更快。

**12. 压缩CSS代码**

不管你在页面中如何使用CSS，CSS文件都是越小越好，这会帮助你提升网页的加载速度。你可以通过 Minify CSS工具 来压缩你的CSS代码。

压缩前：

**[css]** view plaincopy

1. body
2. {
3. **background-color**:#d0e4fe;
4. }
5. h1
6. {
7. **color**:orange;
8. **text-align**:center;
9. }

压缩后：

**[css]** view plaincopy

1. body {**background-color**:#d0e4fe;}
2. h1 {**color**:orange;**text-align**:center;}

**13. 尽量减少DNS查询次数**

当浏览器与Web服务器建立连接时，它需要进行DNS解析，将域名解析为IP地址。然而，一旦客户端需要执行DNS lookup时，等待时间将会取决于域名服务器的有效响应的速度。

虽然所有的ISP的DNS服务器都能缓存域名和IP地址映射表，但如果缓存的DNS记录过期了而需要更新，则可能需要通过遍历多个DNS节点，有时候需要通过全球范围内来找到可信任的域名服务器。一旦域名服务器工作繁忙，请求解析时就需要排队，则进一步延迟等待时间。

因此，减少DNS的查询次数非常重要，页面加载时就尽量避免额外耗时。为了减少DNS查询次数，最好的解决方法就是在页面中减少不同的域名请求的机会。

你可以通过 request checker工具 来检测页面中存在多少请求，然后进行优化。

**14. 尽量减少重定向**

有时为了特定需求，需要在网页中使用重定向。重定向的意思是，用户的原始请求（例如请求A）被重定向到其他的请求（例如请求B）。

但是这会造成网站性能和速度下降，因为浏览器访问网址是一连串的过程，如果访问到一半而跳到新地址，就会重复发起一连串的过程，这将浪费很多的时间。所以我们要尽量避免重定向，建议：

* 不要链接到一个包含重定向的页面
* 不要请求包含重定向的资源

**15. 优化样式表和脚本的顺序**

Style标签和样式表调用代码应该放置在JavaScript代码的前面，这样可以使页面的加载速度加快。

**[html]** view plaincopy

1. **<head>**
2. **<meta** name=description content=”description”**/>**
3. **<title>**title**</title>**
4. **<style>**
5. page specific css code goes here
6. **</style>**
7. **<script** type=”text/javascript”**>**
8. javascript code goes here
9. **</script>**
10. **</head>**

**16. 避免JavaScripts阻塞渲染**

浏览器在遇到一个引入外部JS文件的<script>标签时，会停下所有工作来下载并解析执行它，在这个过程中，页面渲染和用户交互完全被阻塞了。这时页面加载就会停止。

谷歌 建议 删除干扰页面中第一屏内容加载的JavaScript，第一屏是指用户在屏幕中最初看到的页面，无论是桌面浏览器、手机，还是平板电脑。

**17. 缩小原始图像**

如果无需在页面中显示较大的图像，那么就建议将图像的实际大小缩小为显示的大小，这样可以减少下载图像所需的时间。

**18. 指定图像尺寸**

当浏览器加载页面的HTML代码时，有时候需要在图片下载完成前就对页面布局进行定位。如果HTML里的图片没有指定尺寸（宽和高），或者代码描述的尺寸与实际图片的尺寸不符时，浏览器则要在图片下载完成后再“回溯”该图片并重新显示，这将消耗额外的时间）。

所以，最好为页面中的每一张图片都指定尺寸，不管是在HTML里的<img>标签中，还是在CSS中。

### YSlow Web优化最佳实践

**1. 内容优化**

* 尽量减少HTTP请求：常见方法包括合并多个CSS文件和JavaScript文件，利用CSS Sprites整合图像，Image map（图像中不同的区域设置不同的链接），内联图象（使用 data: URL scheme 在实际的页面嵌入图像数据）等。
* 减少DNS查找
* 避免重定向
* 使Ajax可缓存
* 延迟加载组件：考虑哪些内容是页面呈现时所必需首先加载的、哪些内容和结构可以稍后再加载，根据这个优先级进行设定。
* 预加载组件：预加载是在浏览器空闲时请求将来可能会用到的页面内容（如图像、样式表和脚本）。当用户要访问下一个页面时，页面中的内容大部分已经加载到缓存中了，因此可以大大改善访问速度。
* 减少DOM元素数量：页面中存在大量DOM 元素，会导致JavaScript遍历DOM的效率变慢。
* 根据域名划分页面内容：把页面内容划分成若干部分可以使你最大限度地实现平行下载。但要确保你使用的域名数量在2个到4个之间（否则与第2条冲突）。
* 最小化iframe的数量：iframes 提供了一个简单的方式把一个网站的内容嵌入到另一个网站中。但其创建速度比其他包括JavaScript和CSS的DOM元素的创建慢了1-2个数量级。
* 避免404：HTTP请求时间消耗是很大的，因此使用HTTP请求来获得一个没有用处的响应（例如404没有找到页面）是完全没有必要的，它只会降低用户体验而不会有一点好处。

**2. 服务器优化**

* 使用内容分发网络（CDN）：把你的网站内容分散到多个、处于不同地域位置的服务器上可以加快下载速度。
* 添加Expires或Cache-Control信息头：对于静态内容，可设置文件头过期时间Expires的值为“Never expire（永不过期）”；对于动态内容，可使用恰当的Cache-Control文件头来帮助浏览器进行有条件的请求。
* Gzip压缩
* 设置ETag：ETags（Entity tags，实体标签）是web服务器和浏览器用于判断浏览器缓存中的内容和服务器中的原始内容是否匹配的一种机制。
* 提前刷新缓冲区：当用户请求一个页面时，服务器会花费200到500毫秒用于后台组织HTML文件。在这期间，浏览器会一直空闲等待数据返回。在PHP中，可以使用flush()方法，它允许你把已经编译的好的部分HTML响应文件先发送给浏览器，这时浏览器就会可以下载文件中的内容（脚本等）而后台同时处理剩余的HTML页面。
* 对Ajax请求使用GET方法：当使用XMLHttpRequest时，浏览器中的POST方法会首先发送文件头，然后才发送数据。因此使用GET最为恰当。
* 避免空的图像src

**3. Cookie优化**

* 减小cookie大小：去除不必要的coockie，并使coockie体积尽量小以减少对用户响应的影响
* 针对Web组件使用域名无关的Cookie：对静态组件的Cookie读取是一种浪费，使用另一个无Cookie的域名来存放静态组件是一个好方法，或者也可以在Cookie中只存放带www的域名。

**4. CSS优化**

* 将CSS代码放在HTML页面的顶部
* 避免使用CSS表达式：CSS表达式在执行时候的运算量非常大，会对页面性能产生大的影响
* 使用<link>来代替@import
* 避免使用Filters：IE独有属性AlphaImageLoader用于修正IE 7以下版本中PNG图片的半透明效果，但它的问题在于浏览器加载图片时它会终止内容的呈现并且冻结浏览器。

**5. JavaScript优化**

* 将JavaScript脚本放在页面的底部
* 将JavaScript和CSS作为外部文件来引用：在实际应用中使用外部文件可以提高页面速度，因为JavaScript和CSS文件都能在浏览器中产生缓存。
* 缩小JavaScript和CSS
* 删除重复的脚本
* 最小化DOM的访问：使用JavaScript访问DOM元素比较慢
* 开发智能的事件处理程序

**6. 图像优化**

* 优化图片大小
* 通过CSS Sprites优化图片
* 不要在HTML中使用缩放图片
* favicon.ico要小而且可缓存

**7. 针对移动优化**

* 保持组件大小在25KB以下：主要是因为iPhone不能缓存大于25K的文件（注意这里指的是解压缩后的大小）。
* 将组件打包成为一个复合文档：把页面内容打包成复合文本就如同带有多附件的Email，它能够使你在一个HTTP请求中获取多个组件。

### Web性能优化工具



**1. Google PageSpeed**

Google提供了 PageSpeed工具，这是一个浏览器插件，可以很好地应用上文中Google所提到的Web优化实践——帮助你轻松对网站的性能瓶颈进行分析，并为你提供优化建议。

* 在线分析你的网站
* 安装浏览器插件（ Chrome、 Firefox）
* 通过 Insights API在应用中嵌入PageSpeed功能

**2. 雅虎 YSlow**

YSlow是雅虎推出的一款浏览器插件，可以帮助你对网站的页面进行分析，并为你提供一些优化建议，以提高网站的性能。

* Firefox插件
* Chrome插件
* YSlow for Mobile/Bookmarklet
* 源码

**3. 其他分析优化工具**

* 蜘蛛模拟器：这个工具可以分析你的页面，并提供一些优化建议。
* 图像SEO工具：这个工具可以检查图片的alt标签，并提供一些优化建议。
* 请求检查器：找出页面中需要加载哪些资源和服务。
* 链接检查器：检查页面中内部、外部和无效链接。
* HTTP头检查：显示网页或资源的HTTP响应头。
* 社交检查器：检查页面中的社交组件，比如Google+、Facebook、Twitter、Linkedin和Pinterest。
* If modified检查器：检查页面是否接受 If-Modified-Since HTTP头。
* Gzip检查器：检查页面是否经过了Gzip压缩。
* CSS delivery工具：检查页面中所使用的CSS文件。
* 面包屑工具：可根据你输入的信息提供面包屑导航的代码。
* CSS压缩工具：用于压缩CSS代码。

通过以上的优化建议和优化工具，可以轻松找到影响你的Web页面性能的瓶颈，轻松实现Web页面性能的提升。

## 负载均衡的原理、分类、实现架构，以及使用场景

### 为什么需要负载均衡

当系统面临大量用户访问，负载过高的时候，通常会使用增加服务器数量来进行横向扩展，使用集群和负载均衡提高整个系统的处理能力。

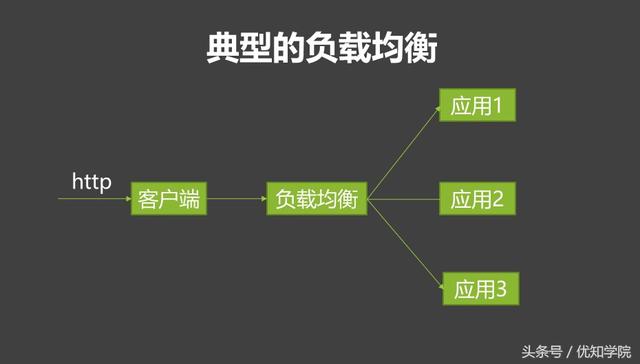
从单机网站到分布式网站，很重要的区别是业务拆分和分布式部署，将应用拆分后，部署到不同的机器上，实现大规模分布式系统。分布式和业务拆分解决了，从集中到分布的问题，但是每个部署的独立业务还存在单点的问题和访问统一入口问题，为解决单点故障，我们可以采取冗余的方式。将相同的应用部署到多台机器上。解决访问统一入口问题，我们可以在集群前面增加负载均衡设备，实现流量分发。

### 负载均衡的原理

系统的扩展可分为纵向（垂直）扩展和横向（水平）扩展。

纵向扩展，是从单机的角度通过增加硬件处理能力，比如CPU处理能力，内存容量，磁盘等方面，实现服务器处理能力的提升，不能满足大型分布式系统（网站），大流量，高并发，海量数据的问题。

因此需要采用横向扩展的方式，通过添加机器来满足大型网站服务的处理能力。比如：一台机器不能满足，则增加两台或者多台机器，共同承担访问压力。这就是典型的集群和负载均衡架构：如下图：



* 应用集群：将同一应用部署到多台机器上，组成处理集群，接收负载均衡设备分发的请求，进行处理，并返回相应数据。
* 负载均衡设备：将用户访问的请求，根据负载均衡算法，分发到集群中的一台处理服务器。（一种把网络请求分散到一个服务器集群中的可用服务器上去的设备）

### 负载均衡的作用

1.解决并发压力，提高应用处理性能（增加吞吐量，加强网络处理能力）；

2.提供故障转移，实现高可用；

3.通过添加或减少服务器数量，提供网站伸缩性（扩展性）；

4.安全防护；（负载均衡设备上做一些过滤，黑白名单等处理）

### 负载均衡的分类



1）二层负载均衡（mac）

根据OSI模型分的二层负载，一般是用虚拟mac地址方式，外部对虚拟MAC地址请求，负载均衡接收后分配后端实际的MAC地址响应）

2）三层负载均衡（ip）

一般采用虚拟IP地址方式，外部对虚拟的ip地址请求，负载均衡接收后分配后端实际的IP地址响应）

3）四层负载均衡（tcp）

在三次负载均衡的基础上，用ip+port接收请求，再转发到对应的机器。

4）七层负载均衡（http）

根据虚拟的url或IP，主机名接收请求，再转向相应的处理服务器。

### 最常见的四层和七层负载均衡

1）四层的负载均衡就是基于IP+端口的负载均衡：在三层负载均衡的基础上，通过发布三层的IP地址（VIP），然后加四层的端口号，来决定哪些流量需要做负载均衡。

对应的负载均衡器称为四层交换机（L4 switch），主要分析IP层及TCP/UDP层，实现四层负载均衡。此种负载均衡器不理解应用协议（如HTTP/FTP/MySQL等等）。

**实现四层负载均衡的软件有：**

* F5：硬件负载均衡器，功能很好，但是成本很高。
* lvs：重量级的四层负载软件
* nginx：轻量级的四层负载软件，带缓存功能，正则表达式较灵活
* haproxy：模拟四层转发，较灵活

2）七层的负载均衡就是基于虚拟的URL或主机IP的负载均衡

对应的负载均衡器称为七层交换机（L7 switch），除了支持四层负载均衡以外，还有分析应用层的信息，如HTTP协议URI或Cookie信息，实现七层负载均衡。此种负载均衡器能理解应用协议。

**实现七层负载均衡的软件有：**

* haproxy：天生负载均衡技能，全面支持七层代理，会话保持，标记，路径转移；
* nginx：只在http协议和mail协议上功能比较好，性能与haproxy差不多；
* apache：功能较差
* Mysql proxy：功能尚可。

总的来说，一般是lvs做4层负载；nginx做7层负载；haproxy比较灵活，4层和7层负载均衡都能做。

### 负载均衡应用场景



**场景一：应用于高访问量的业务**

如果应用访问量很高，可以通过配置监听规则将流量分发到不同的服务器上。

**场景二：横向扩张系统**

可以根据业务发展的需要，通过随时添加和移除服务器，来扩展应用系统的服务能力，适用于各种Web服务器和App服务器。

**场景三：消除单点故障**

当其中一部分服务器发生故障后，负载均衡会自动屏蔽故障的服务器，将请求分发给正常运行的服务器，保证应用系统仍能正常工作。

**场景四：同城容灾 （多可用区容灾）**

为了提供更加稳定可靠的负载均衡服务，当主可用区出现机房故障或不可用时，负载均衡仍然有能力在非常短的时间内切换到另外一个备可用区恢复服务能力；当主可用区恢复时，负载均衡同样会自动切换到主可用区提供服务，保证服务依然正常运行。

## 常见分布式文件存储介绍、选型比较、架构设计

### 数据正成为世界上最有价值的资源，分布式文件存储是应对数据爆炸的最好解决方案，那就会涉及到分布式文件存储方案、选型、架构设计等。

### 分布式文件存储的来源

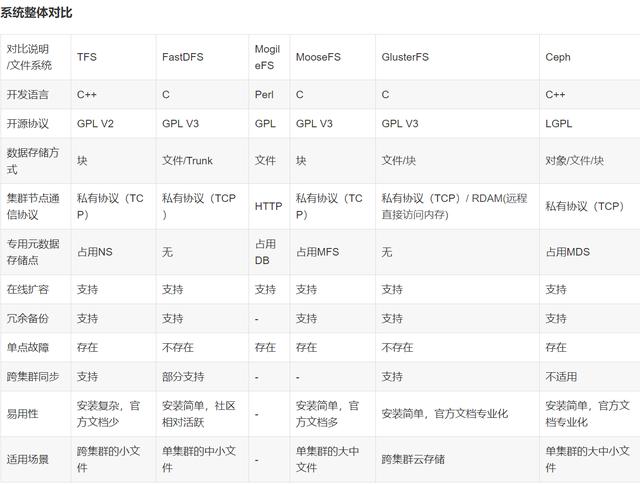
在这个数据爆炸的时代，产生的数据量不断地在攀升，从GB,TB,PB,ZB.挖掘其中数据的价值也是企业在不断地追求的终极目标。但是要想对海量的数据进行挖掘，首先要考虑的就是海量数据的存储问题，比如Tb量级的数据。

谈到数据的存储，则不得不说的是磁盘的数据读写速度问题。早在上个世纪90年代初期，普通硬盘的可以存储的容量大概是1G左右，硬盘的读取速度大概为4.4MB/s.读取一张硬盘大概需要5分钟时间，但是如今硬盘的容量都在1TB左右了,相比扩展了近千倍。但是硬盘的读取速度大概是100MB/s。读完一个硬盘所需要的时间大概是2.5个小时。所以如果是基于TB级别的数据进行分析的话，光硬盘读取完数据都要好几天了，更谈不上计算分析了。那么该如何处理大数据的存储，计算分析呢？

### 常见的分布式文件系统

GFS、HDFS、Lustre 、Ceph 、GridFS 、mogileFS、TFS、FastDFS等。各自适用于不同的领域。它们都不是系统级的分布式文件系统，而是应用级的分布式文件存 储服务。

**分布式文件存储选型比较**



### 知名开源分布式文件存储

**1.GFS（Google File System）**

Google公司为了满足本公司需求而开发的基于Linux的专有分布式文件系统。尽管Google公布了该系统的一些技术细节，但Google并没有将该系统的软件部分作为开源软件发布。

**2.HDFS**

Hadoop 实现了一个分布式文件系统（Hadoop Distributed File System），简称HDFS。 Hadoop是Apache Lucene创始人Doug Cutting开发的使用广泛的文本搜索库。它起源于Apache Nutch，

后者是一个开源的网络搜索引擎，本身也是Luene项目的一部分。Aapche Hadoop架构是MapReduce算法的一种开源应用，是Google开创其帝国的重要基石。

**3.TFS**

TFS（Taobao FileSystem）是一个高可扩展、高可用、高性能、面向互联网服务的分布式文件系统，主要针对海量的非结构化数据，它构筑在普通的Linux机器 集群上，可为外部提供高可靠

和高并发的存储访问。TFS为淘宝提供海量小文件存储，通常文件大小不超过1M，满足了淘宝对小文件存储的需求，被广泛地应用 在淘宝各项应用中。它采用了HA架构和平滑扩容，保证了整个文件系统的可用性和扩展性。同时扁平化的数据组织结构，可将文件名映射到文件的物理地址，简化 了文件的访问流程，一定程度上为TFS提供了良好的读写性能。

Google学术论文，这是众多分布式文件系统的起源，**HDFS和TFS都是参考Google的GFS设计出来的。**

### 典型的分布式文件存储的架构设计

我以hadoop的HDFS为例，毕竟开源的分布式文件存储使用的最多。

Hadoop分布式文件系统(HDFS)被设计成适合运行在通用硬件(commodity hardware)上的分布式文件系统。HDFS是一个高度容错性的系统，适合部署在廉价的机器上。HDFS能提供高吞吐量的数据访问，非常适合大规模数据集上的应用。HDFS放宽了一部分POSIX约束，来实现流式读取文件系统数据的目的。

**大规模数据集**

运行在HDFS上的应用具有很大的数据集。HDFS上的一个典型文件大小一般都在G字节至T字节。因此，HDFS被调节以支持大文件存储。它应该能提供整体上高的数据传输带宽，能在一个集群里扩展到数百个节点。一个单一的HDFS实例应该能支撑数以千万计的文件。

**简单的一致性模型**

HDFS应用需要一个“一次写入多次读取”的文件访问模型。一个文件经过创建、写入和关闭之后就不需要改变。这一假设简化了数据一致性问题，并且使高吞吐量的数据访问成为可能。Map/Reduce应用或者网络爬虫应用都非常适合这个模型。目前还有计划在将来扩充这个模型，使之支持文件的附加写操作。

**异构软硬件平台间的可移植性**

HDFS在设计的时候就考虑到平台的可移植性。这种特性方便了HDFS作为大规模数据应用平台的推广。

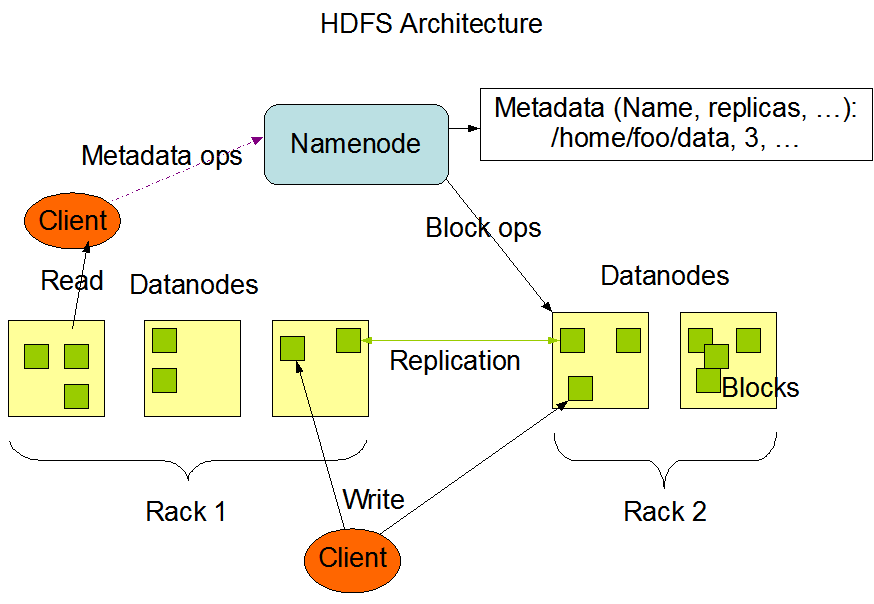
**Namenode 和 Datanode**

HDFS采用master/slave架构。一个HDFS集群是由一个Namenode和一定数目的Datanodes组成。

Namenode是一个中心服务器，负责管理文件系统的名字空间(namespace)以及客户端对文件的访问。

集群中的Datanode一般是一个节点一个，负责管理它所在节点上的存储。HDFS暴露了文件系统的名字空间，用户能够以文件的形式在上面存储数据。从内部看，一个文件其实被分成一个或多个数据块，这些块存储在一组Datanode上。

Namenode执行文件系统的名字空间操作，比如打开、关闭、重命名文件或目录。它也负责确定数据块到具体Datanode节点的映射。Datanode负责处理文件系统客户端的读写请求。在Namenode的统一调度下进行数据块的创建、删除和复制。



Namenode和Datanode被设计成可以在普通的商用机器上运行。这些机器一般运行着GNU/Linux操作系统(OS)。HDFS采用Java语言开发，因此任何支持Java的机器都可以部署Namenode或Datanode。由于采用了可移植性极强的Java语言，使得HDFS可以部署到多种类型的机器上。一个典型的部署场景是一台机器上只运行一个Namenode实例，而集群中的其它机器分别运行一个Datanode实例。这种架构并不排斥在一台机器上运行多个Datanode，只不过这样的情况比较少见。

### 分布式存储的未来

随着现代社会从工业时代过渡到信息时代，信息技术的发展以及人类生活的智能化带来数据的爆炸性增长，**数据正成为世界上最有价值的资源。**

根据物理存储形态，数据存储可分为集中式存储与分布式存储两种。集中式存储以传统存储阵列（传统存储）为主，分布式存储（云存储）以软件定义存储为主。

**传统存储**一向以可靠性高、稳定性好，功能丰富而著称，但与此同时，传统存储也暴露出横向扩展性差、价格昂贵、数据连通困难等不足，容易形成数据孤岛，导致数据中心管理和维护成本居高不下。

**分布式存储：**将数据分散存储在网络上的多台独立设备上，一般采用标准x86服务器和网络互联，并在其上运行相关存储软件，系统对外作为一个整体提供存储服务。。

总之，分布式文件存储，不仅提高了存储空间的利用率，还实现了弹性扩展，降低了运营成本，避免了资源浪费，更适合未来的数据爆炸时代场景。

## 开源搜索引擎Lucene、Solr、Sphinx等优劣势比较

### 开源搜索引擎分类

1.Lucene系搜索引擎，java开发,包括：

* Lucene
* Solr
* Elasticsearch
* Katta、Compass等都是基于Lucene封装。

你可以想象Lucene系有多强大。

2.Sphinx搜素引擎，c++开发,简单高性能。

**以下重点介绍最常用的开源搜素引擎：Lucene、Solr、Elasticsearch、Sphinx的特点和优劣势选型比较。**

### Lucene

**1.Lucene简介**

Lucene的开发语言是Java，也是Java家族中最为出名的一个开源搜索引擎，在Java世界中已经是标准的全文检索程序，它提供了完整的查询引擎和索引引擎，没有中文分词引擎，需要自己去实现，因此用Lucene去做一个搜素引擎需要自己去架构，另外它不支持实时搜索。但是solr和elasticsearch都是基于Lucene封装。

**2.Lucene的优劣势**

**优点：**

成熟的解决方案，有很多的成功案例。apache 顶级项目，正在持续快速的进步。庞大而活跃的开发社区，大量的开发人员。它只是一个类库，有足够的定制和优化空间：经过简单定制，就可以满足绝大部分常见的需求；经过优化，可以支持 10亿+ 量级的搜索。

**缺点：**

需要额外的开发工作。所有的扩展，分布式，可靠性等都需要自己实现；非实时，从建索引到可以搜索中间有一个时间延迟，而当前的“近实时”(Lucene Near Real Time search)搜索方案的可扩展性有待进一步完善

### Apache Solr



**1.Slor简介**

Solr是一个高性能，采用Java开发，基于Lucene的全文搜索服务器。

文档通过Http利用XML加到一个搜索集合中。

查询该集合也是通过 http收到一个XML/JSON响应来实现。它的主要特性包括：高效、灵活的缓存功能，垂直搜索功能，高亮显示搜索结果，通过索引复制来提高可用性，提 供一套强大Data Schema来定义字段，类型和设置文本分析，提供基于Web的管理界面等。

### 2.Solr的优缺点

#### 优点

1. Solr有一个更大、更成熟的用户、开发和贡献者社区。
2. 支持添加多种格式的索引，如：HTML、PDF、微软 Office 系列软件格式以及 JSON、XML、CSV 等纯文本格式。
3. Solr比较成熟、稳定。
4. 不考虑建索引的同时进行搜索，速度更快。

#### 缺点

1. 建立索引时，搜索效率下降，实时索引搜索效率不高。

### Elastic Search



**1.ElasticSearch简介**

ElasticSearch是一个基于Lucene构建的开源，分布式，RESTful搜索引擎。设计用于云计算中，能够达到实时搜索，稳定，可靠，快速，安装使用方便。支持通过HTTP使用JSON进行数据索引。

**2.Elasticsearch的优缺点**

#### 优点

1. Elasticsearch是分布式的。不需要其他组件，分发是实时的，被叫做”Push replication”。
2. Elasticsearch 完全支持 Apache Lucene 的接近实时的搜索。
3. 处理多租户（multitenancy）不需要特殊配置，而Solr则需要更多的高级设置。
4. Elasticsearch 采用 Gateway 的概念，使得完备份更加简单。
5. 各节点组成对等的网络结构，某些节点出现故障时会自动分配其他节点代替其进行工作。

#### 缺点

1. 还不够自动（不适合当前新的Index Warmup API）

### Elasticsearch 与 Solr 的比较总结

* 二者安装都很简单；
* Solr 利用 Zookeeper 进行分布式管理，而 Elasticsearch 自身带有分布式协调管理功能;
* Solr 支持更多格式的数据，而 Elasticsearch 仅支持json文件格式；
* Solr 官方提供的功能更多，而 Elasticsearch 本身更注重于核心功能，高级功能多有第三方插件提供；
* Solr 在传统的搜索应用中表现好于 Elasticsearch，但在处理实时搜索应用时效率明显低于 Elasticsearch。

总之，Solr 是传统搜索应用的有力解决方案，但 Elasticsearch 更适用于新兴的实时搜索应用。

### Sphinx



#### 1.Sphinx简介

Sphinx一个基于SQL的全文检索引擎，特别为一些脚本语言（PHP,Python，Perl，Ruby）设计搜索API接口。

Sphinx是一个用C++语言写的开源搜索引擎，也是现在比较主流的搜索引擎之一，在建立索引的事件方面比Lucene快50%，但是索引文件比Lucene要大一倍，因此Sphinx在索引的建立方面是空间换取事件的策略，在检索速度上，和lucene相差不大，但检索精准度方面Lucene要优于Sphinx，另外在加入中文分词引擎难度方面，Lucene要优于Sphinx.其中Sphinx支持实时搜索，使用起来比较简单方便.

Sphinx可以非常容易的与SQL数据库和脚本语言集成。当前系统内置MySQL和PostgreSQL 数据库数据源的支持，也支持从标准输入读取特定格式 的XML数据。通过修改源代码，用户可以自行增加新的数据源（例如：其他类型的DBMS 的原生支持）

**2.Sphinx的特点**

* 高速的建立索引(在当代CPU上，峰值性能可达到10 MB/秒);
* 高性能的搜索(在2 – 4GB 的文本数据上，平均每次检索响应时间小于0.1秒);
* 可处理海量数据(目前已知可以处理超过100 GB的文本数据, 在单一CPU的系统上可 处理100 M 文档);
* 提供了优秀的相关度算法，基于短语相似度和统计（BM25）的复合Ranking方法;
* 支持分布式搜索;
* 支持短语搜索
* 提供文档摘要生成
* 可作为MySQL的存储引擎提供搜索服务;
* 支持布尔、短语、词语相似度等多种检索模式;
* 文档支持多个全文检索字段(最大不超过32个);
* 文档支持多个额外的属性信息(例如：分组信息，时间戳等);
* 支持断词;

## 如何从0到1设计一个类Dubbo的RPC框架

### 之前分享了[如何从0到1设计一个MQ消息队列](http://youzhixueyuan.com/design-the-message-queue.html)，今天谈谈“如何从0到1设计一个Dubbo的RPC框架”，重点考验：

* 你对RPC框架的底层原理掌握程度。
* 以及考验你的整体RPC框架系统设计能力。

### **RPC和RPC框架**

**1.RPC(Remote Procedure Call)**

即远程过程调用, 主要解决远程通信间的问题，不需要了解底层网络的通信机制。

**2.RPC框架**

RPC框架负责屏蔽底层的传输方式（TCP或者UDP）、序列化方式、以及通信细节。

实际使用中，并不需要关心底层通信细节和调用过程，让业务端专注于业务代码的实现。

国内大家熟知的PRC框架，阿里的HSF和**Dubbo(开源)**

### **Dubbo的发展由来**

**1. 业务规模小**

比如早期一个应用Java War包，将所有功能都打包，部署在一个单机服务器，调用接口也比较方便，不涉及到任何分布式场景。



**2.业务规模变大**

随着业务的快速发展，业务越来越多、子系统也越来越多时。比如：淘宝的交易系统、商品系统、用户系统、评价系统…上百个系统的出现。

系统变得越来越复杂，业务代码依然耦合在一起。比如最早期的淘宝denali工程，包含所有业务系统的代码，就仅打包部署都需要很长的时间。

并且，随着每个业务线的快速发展，业务代码耦合在一起，上线后出现问题急需要回滚代码，拉分支、大量的代码merge工作，这个过程极其痛苦。

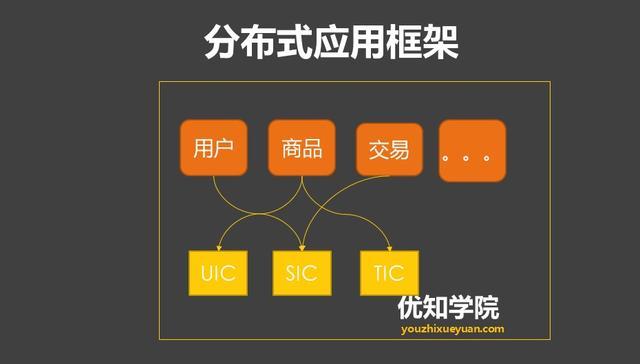
这个时候，你会发现技术已经成了业务的瓶颈，急需把业务单独抽离出来，各自单独部署。

**3.Dubbo和HSF的出现**

应用系统一旦涉及到拆分部署，问题就来了，急需一种高效的应用程序间的通讯手段来完成这种需求，这就会涉及到**分布式远程调用**。

于是，淘宝就把denali按照业务为单位拆分成了类似这样的系统：UM(UserManger)、SM(ShopManager)..等等几十个工程代码。

再按照业务为单位，把所有调用相关的接口以业务为单元进行拆分：UIC(用户中心服务)、SIC(店铺中心服务)…等等以业务为单位集群部署，按照业务提供服务。



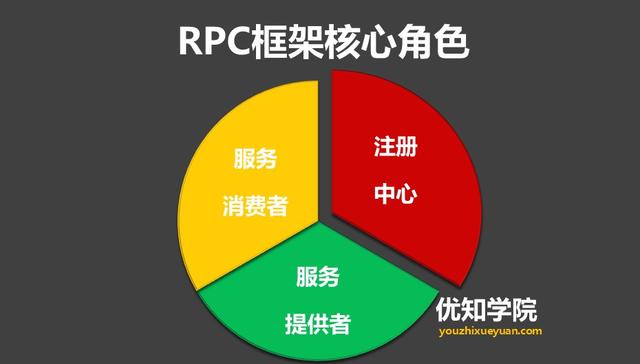
所以，RPC的框架来了，阿里内部使用HSF，以及开源的RPC 框架：Dubbo。

### **RPC框架的核心设计**

前面优知mikechen提到了RPC的核心目标：主要是解决分布式系统中服务之间的调用问题。

其实，走到这一步涉及的知识体系非常的多：要求对通信、远程调用、消息机制等有深入的理解和掌握，要求的都是从理论、硬件级、操作系统级以及所采用的语言的实现都有清楚的理解。

**1.RPC框架三个核心角色**



**1)服务提供者（Server）**

对外提供后台服务，将自己的服务信息，注册到注册中心

**2)注册中心（Registry）**

用于服务端注册远程服务以及客户端发现服务。

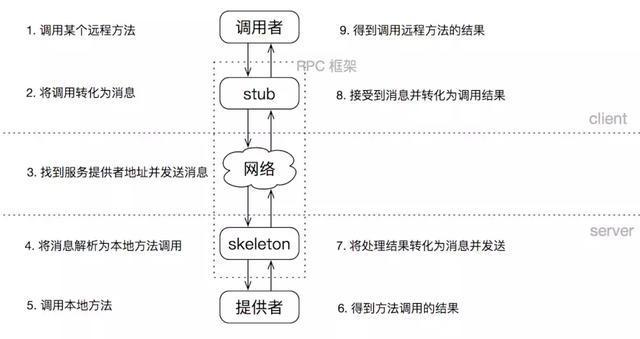
目前主要的注册中心可以借由 zookeeper，eureka，consul，etcd 等开源框架实现。

比如：阿里的Dubbo就是采用zookeeper实现注册中心。

**3)服务消费者（Client）**

从注册中心获取远程服务的注册信息，然后进行远程过程调用。

**2.RPC远程调用过程**



1）服务调用方（client）调用以本地调用方式调用服务；

2）client stub接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体；在Java里就是序列化的过程

3）client stub找到服务地址，并将消息通过网络发送到服务端；

4）server stub收到消息后进行解码,在Java里就是反序列化的过程；

5）server stub根据解码结果调用本地的服务；

6）本地服务执行处理逻辑；

7）本地服务将结果返回给server stub；

8）server stub将返回结果打包成消息，Java里的序列化；

9）server stub将打包后的消息通过网络并发送至消费方

10）client stub接收到消息，并进行解码, Java里的反序列化；

11）服务调用方（client）得到最终结果。

**RPC框架的目标就是要2~10这些步骤都封装起来。**

### **RPC框架涉及技术**

**1.建立通信**

首先，要解决通讯的问题，主要是通过在客户端和服务器之间建立TCP连接，远程过程调用的所有交换的数据都在这个连接里传输。

**2.服务寻址**

**1）服务注册**

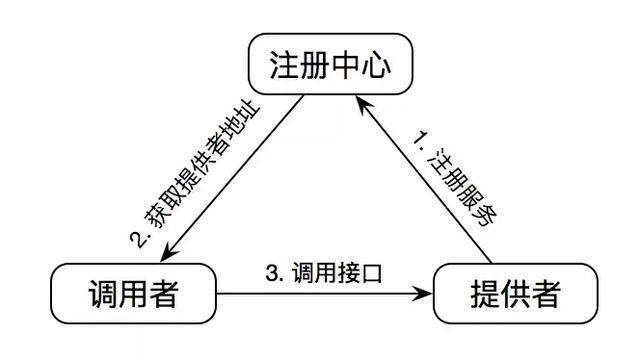
首先需要把服务注册到服务中心。其实就是在注册中心进行一个登记，注册中心存储了该服务的IP、端口、调用方式(协议、序列化方式)等。在zookeeper中，进行服务注册，实际上就是在zookeeper中创建了一个znode节点，该节点存储了上面所说的服务信息。

**2）服务发现**

服务消费者在第一次调用服务时，会通过注册中心找到相应的服务的IP地址列表，并缓存到本地，以供后续使用。当消费者调用服务时，不会再去请求注册中心，而是直接通过负载均衡算法从IP列表中取一个服务提供者的服务器调用服务。

**3）注册服务**

可靠的寻址方式（主要是提供服务的发现）是RPC的实现基石，比如可以zookeeper来实现注册服务等等。



* 服务提供者启动后主动向服务（注册）中心注册机器ip、端口以及提供的服务列表。
* 服务消费者启动时向服务（注册）中心获取服务提供方地址列表，可实现软负载均衡和Failover。
* 提供者需要定时向注册中心发送心跳，一段时间未收到来自提供者的心跳后，认为提供者已经停止服务，从注册中心上摘取掉对应的服务等等。

**3.网络传输**

数据传输采用什么协议，数据该如何序列化和反序列化

**4.NIO通信**

当前很多RPC框架都直接基于netty这一IO通信框架，比如阿里巴巴的HSF、dubbo，Hadoop Avro，推荐使用Netty 作为底层通信框架。

5.**服务调用**

比如：B机器进行本地调用（通过代理Proxy）之后得到了返回值，此时还需要再把返回值发送回A机器，同样也需要经过序列化操作，然后再经过网络传输将二进制数据发送回A机器，而当A机器接收到这些返回值之后，则再次进行反序列化操作

总之，要实现一个RPC不算难，难的是实现一个高性能高可靠的RPC框架**，**后续将剖析Dubbo，看看Dubbo是如何来解决这些难题。

# 职场发展

## 互联网寒冬，程序员如何突破重围？我的3个建议

2018年，互联网寒冬、裁员成了我们的热词。

2019年，大佬们预言，没有最冷，只有更冷。

最近这两个月，有些小伙伴通过私信或评论给我留言：

* 工作几年了，感觉技术到了瓶颈期，怎么突破？
* 一线程序员该如何应对中年危机？
* 现在的工作得不到成长，跳槽又担心经济寒冬工作不好找，有什么建议吗？

……

大家纷纷表达着对未来的不确定，我感受到了技术人的深深焦虑。

追本溯源，这种不确定和焦虑主要还是源于大家对能力与未来的一种不自信。

怎样破解这种不自信？如果大家未来不想被贴上中年危机的标签，不想做经济寒冬下的被动择业者，不想被互联网变革浪潮冲出赛道。那么你需要：**尽可能早播种（明确职业发展目标）、坚持学习精进，去建立自己的核心竞争力**。

我根据每个人的不同阶段，给到以下3个建议：

### **23-27岁：技术积累阶段**

假设本科23岁毕业，那么工作的前5年对你来说就是打基础的阶段。

在这5年时间里面，你要积累足够的技术底子，打磨自己的技术实力，**成为某一个技术细分领域的牛人**。

### **27-30岁：形成思维方法论和知识体系的阶段**

当你积累足够的技术实力，例如超过10万行代码以后，你应该形成了自己的思维方法论和自己独立的学习技巧，任何新的技术在你眼中都能迅速的看到技术的本质，快速吸收成为你的知识体系的一部分。

在这个阶段，积累技术对你来说简直是小菜一碟，**你更需要磨练的是思考能力**，**形成自己的思维方法和知识体系。**

这里的知识体系除了[架构图谱](http://mp.weixin.qq.com/s?__biz=MzI0NzU3MTk2MQ==&mid=2247485916&idx=1&sn=797d937dbb8295ee13aa5aead6ef8672&chksm=e9acbd4adedb345cfe40a36ac61a6b672949d97111e4d60fcc4dbbe2041a294aa42fd83de445&scene=21#wechat_redirect)里面谈到的技术体系外，也应该包含产品。

技术服务于什么？产品啊，产品服务于业务，这件事可不能忘了。如果说早期在积累技术的阶段，那这个阶段至少也需要多培养点产品意识，这一点也许会在某一天帮到你。

举一个简单的例子，如果公司裁员，二选一。如果你熟知业务，另外一个伙伴对业务不熟悉，技术实力相当，公司会怎样选择？

以及同样两个伙伴被提升，技术实力相当，公司会怎样选择？那就看谁的价值度，以及不可替代性，公司更原意留住未来能做出更大贡献的。

还有一点，即便你在大公司，架构师不熟悉业务，你怎么服务好产品。

### **30-33岁这个阶段，最关键的是了解你自己**。

知道自己擅长什么，自己的优势在哪里，**怎样才能够最大化自己的优势**。同理也需要知道自己有什么重大劣势**，未来是扬长避短，还是不断补自己的短板。**

这个阶段的选择，也许更考研你的智慧。

其实导致焦虑与缺乏信心的主要原因，我个人认为还是来自于每一个阶段没有坚持做对的事情，技术能力积累不足。

只要你坚持学习进阶，随着工作经验增长、项目经验沉淀、技术精进，你的收入、职位、抗风险能力都将远远高于大多数人，你也将会是生活、事业的赢家。此时技术人**再遇寒冬，相信必能自信笑言：寒冬再冷，我又何惧！**

最后，再送给大家我的一个锦囊：坚持跑步。

## **从职业方向，谈程序员如何突破成长瓶颈？**

**对工作1-3年的程序员来说，依据个人情况提前规划未来职业发展方向，这很重要。**譬如，在未来3-5年甚至更久，工作要做到什么程度？往哪个方向发展?通过怎样的路径才能实现目标？**如果沉浸在当下“程序员高薪”的论调中岁月静好，总归有一天会面临网上热议的“程序员中年危机”，以至于最后不得不步入转行或提前退休的尴尬局面。**

作为已工作十年有余的IT人，我一直认为这个话题对程序员非常重要，今天和大家详细聊聊。

**职业规划的重要性**

**人生和职场就像下象棋，你只有明确了目标才不会在每一步上迷茫，之所有迷茫，是因为不清楚自己最终想要什么，下一个阶段的目标在哪里？**

不久前曾有程序员通过头条号给我私信留言：

某公司表示薪资上浮20%-30%，我应该跳槽加入吗?

我最近收到了几个offer，应该去哪一家更好?

从大公司去小公司,是否值得去尝试？

想成为架构师，但不知道自己的技术是否能胜任？

从程序员转行，但不知道未来自己的竞争力会变弱吗？

……

其实，追求短期利益（眼前）或长期利益（未来），取决于你自己的决定。

我的建议是，不要在你经历之后，再去回顾职业生涯，初期跳槽涨薪确实很快，但到了中后期，跳槽再也带不来前期的涨幅，反而在频繁的跳槽过程中，失去了很多机会，**技能与业务的能力也没能积累沉淀下来**，这将直接导致很多人在即将步入中年时感到异常焦虑，不知道自己未来的路何去何从。

跳槽和转行如果仅仅只是工作逃避的手段，未来你将面临更大的焦虑，未来也根本谈不上什么有什么核心竞争力。但如果跳槽和转行，是谋定而后动，**围绕你的主线职业去发展**，这是完全不一样的结果。

总之，在职业的选择上，绝不能有“**逃避**”二字，逃避将会让你无处藏身，你只有明白了这个道理，你才会去克服眼前的困难，学会承担，这样的人生才有未来。

我在优知学院官网留言区写过一段话，共勉：

我们每个人身上都有自己的使命和责任，或为生活，为事业，为自己内心中那颗躁动不安的心，去努力，去奋斗，去实现可能的一切，而这不啻是一段曼妙的经历，于你，于我！

不用担心生活给予你的任何可能，或成功，或低潮，或平淡，这是所有人都会走过的路！什么样的年龄做什么样的事，什么样的年华承担什么样的责任，一切顺其自然，而这本身就是一种成熟力量！

某年某月，再回首，请记得我们的来时路，**愿初心不改，一如既往！**

**程序员的职业发展方向**

**1、架构师（专注技术，侧重技术）**

每一个程序员都有一个架构师的梦想，这个方向是很多技术人的第一选择，从年薪十几万的程序员成长为年薪几十万甚至过百万的架构师，给力的薪酬已足以让不少程序员心动不已。若未来你想成为架构师，那么现在至少需要清楚了解[架构师的主要职责是什么？](http://youzhixueyuan.com/architects-main-responsibilities.html)，有目标有方向地坚持学习。

**2、项目经理(技术一般，侧重管理)**

往管理方向发展的职业路径，项目经理主要侧重于项目管理工作，主要能力是计划、沟通及组织协调。沟通和表达是开发人员短柄，所以想往此线路发展的同学，从日常的工作、生活中就得逐步加强锻炼这方面能力。

**3、技术经理(技术和管理结合)**

技术经理需要带领团队前行，在国内的技术经理，大部分都在30岁左右。一般，程序员在30岁是一个职业分水岭。

技术经理的这个阶段的工作重点不再是专注于技术，而是在培养团队，带领团队为主。如果你没有让团队的成员变得更好，没有分享，没有利益分享的想法，没有去帮助大家成长的想法，这样的程序员不太适合这个岗位。

总之，这个职位成全大家而不是自己才是重点。

**4、产品经理（转岗）**

如果你对技术发展路径、管理发展路径都不感兴趣，那么不妨考虑转产品经理，移动互联网时代，产品经理的角色越来越重要，一个好的产品经理有可能带给公司颠覆性的发展，收入各方面也还不错，我认识的一些程序员，也有转行做产品经理，并且做得还不错的。

**5、CTO(管理+技术结合，侧重管理)**

需要有一定的技术能力和团队管理能力。CTO的工作更偏业务、偏管理，以及商业等，主要职责包括团队管理、开发流程、公司战略、业务增长以及各类技术疑难问题的解决等，可以说是一个比较全方位的发展方向，一句话，CTO的主要职能是管理+技术+业务+商业的综合发展。

**总结**

不管选择什么样的路径，都离不开提前规划、充分准备，**机会总会留给有准备的人**。程序员是个相对公平的职业，你若盛开，蝴蝶自来，薪酬也会随能力得到明显增幅，那么大家在应对中年危机时更加游刃有余了，这句话送给自己及大家:**Stay hungry ，Stay Foolish。我们，共勉！**

## [技术人如何提升自己的核心竞争力](http://youzhixueyuan.com/how-technical-people-improve-their-core-competitiveness.html)

互联网行业是一个发展非常快，变化也快的行业，在这个行业，总是让人感觉既兴奋又不安。

兴奋的是你总能看到无数新奇的事物，甚至亲身参与到一场变革中去，而不安的则是，任凭你如何NB，你也无法保证哪一天，你就会被无情抛在时代的身后，成了那个被替代的家伙又或是一场新变革的牺牲品。

十年PC时代眨眼就过去了，很快进入到了移动互联网时代，移动互联网时代还未站稳脚跟，很快又将进入人工智能、机器人时代，低效的人工时代很快即将成为过去。

举一个市场的例子：几年前移动互联网时代，APP开发非常火爆，如今很多APP的开发市场需求在急剧减少，之前比较吃香的IOS等开发者，如今也没有之前这么风光了。

市场的变化莫测，不得不让人深思，作为个人在互联网行业，如何才能不断提升自己的核心竞争力，这也一直都是一个大家比较关心的话题。

我个人认为30岁之前尽量专业发展，30岁之后横向扩展是主体方向。

**1.23-27岁：技术积累阶段**

这个阶段的关键点就是打磨自己的技术实力，成为某一个技术细分领域的牛人，这是一个考核的标准。

**2.27-30岁：形成自己思维方法论和建立完善的技能体系。**

比如能掌握完整的技能体系框架，以及培养自己的独立思维能力，对待工作以及生活有自己的独立见解，而不是随波逐流。

**30-33岁  开始横向扩展阶段**

掌握完整的技能以及充分培养了自己的独立思考，甚至自学的能力，开始懂得自己擅长什么，自己的核心优势在哪里(核心的思维竞争力，不仅仅只是技能)，怎样才能够最大化自己的价值。

30岁之后是在公司继续待下去，短期继续稳扎稳打，工作和收入也都很稳定，还是做眼长期，以空间换时间，跳出现阶段的温室区，寻求自我突破的机会，以及寻求好的机遇加入创业公司，从何进一步提升自己的核心竞争力？

[写给30岁后想创业的朋友](http://youzhixueyuan.com/write-to-a-friend-who-wants-to-start-a-business.html)，这篇文章里有我详细的思考和建议。

今天再补充几点思考：

**1 传统企业背景的创业公司，请慎重考虑**

这一类比例最高，很多做技术的朋友，做到管理层这个职位的，对市场、营销缺乏足够的了解，也对企业、老板、行业，认识和分析得不够透彻，很多加入的技术朋友，大概率都成为了炮灰。

传统企业转型的公司，对于互联网的认识是不够的，大部分都是短期投资为导向，对于你的考核基本也就在3个月，往往口头承若这件事长期投入，短期其实就是3个月为结果导向，这一点你需要考量老板转型的决心(思维往往非常难以转换)。

如果你一定要加入，建议对公司、行业地位、老板做好充分的分析和调研，不要仅仅考量薪资，这是我对于加入这样创业公司的建议。

**2 创业不等于财务自由，更不等于快乐工作**

从财务的角度，你对公司有价值，有股票或期权,要么是早期前20,要么是C轮以后进入的管理层,如果按照所有加入创业公司的人的比例来看，最终能财务自由的人都不到1%。

**3.牛逼的背后是大量的苦逼(往往被大部分人忽略)**

很多成功人士，你看见他牛逼的地方，只是他其中很少一部分外表光鲜的部分，他之前大量苦逼的经历，以及彷徨迷茫的阶段，都被光芒所隐藏起来了，每个人牛逼的人都是这样走过来的。这给大部分人带了幻觉，觉得成功并不难啊。

**书不然，真正牛逼的人背后都是没日没夜的苦逼来支撑。**

**什么样的人适合走出公司创业？**

**1.有巨大野心和执着想要获得创业成功的。**

创业就是一心为了寻求些新体验，将之视为另一种成长路径，完全可以坦然接受失败的，这样心态会好很多。（30岁以下最佳，30岁以上仍然野心未灭）

**2.对于某件事有极强的信念和执着，觉得这件事自己非做不可，即便为此付出一切也值得的。**

**3.自己多年隐忍积累攒了一手好牌，自觉已经有了充裕的资源、人脉和能力储备的，做好准备要大干一场的。**

## **又一名华为中年程序员倒下了，如何打破35岁程序员的魔咒？**

昨天上午，微博和脉脉网友爆料称，前天上午6点多，坂田天安云谷星一男子跳楼身亡，经进一步证实，该名男子为华为外包程序员，跳楼原因为绩效被黑。

不知从何时起，很多企业将入职门槛限定在35岁以下，“35岁”已然成为职场老鸟的魔咒，而近年来有关“程序员猝死”、“程序员跳楼”等新闻层出不穷，这股百般苦恼的邪气，已入侵到了程序员群体，每每看到这类信息，我的心情都无比沉重，刚好我今年35岁，正经历着大多数人在这个年龄所面临的困惑、承担的压力、肩上的重担，这个年龄的遭遇我都感同深受，希望借此篇文章与大家一起分享心情、心得。

**1、 近年非正常死亡事件回顾**

2016年6月23日，阿里数据技术及产品部总监欧吉良打羽毛球猝死；

2015年12月13日，腾讯互娱的员工原腾讯互娱技术研发中心语音引擎组副组长李俊明，散步时猝死；

2015年10月5日，年仅33岁的游戏企业广州仙海网络公司总裁张旭过度劳累引发突发心脏病逝世；

2015年3月24日，年仅36岁的深圳闻泰通讯IT男张斌被发现猝死在居住所马桶上；

2013年7月15日，年仅36岁的淘宝电商淘品牌御泥坊前董事长吴立君突发脑疾逝世；

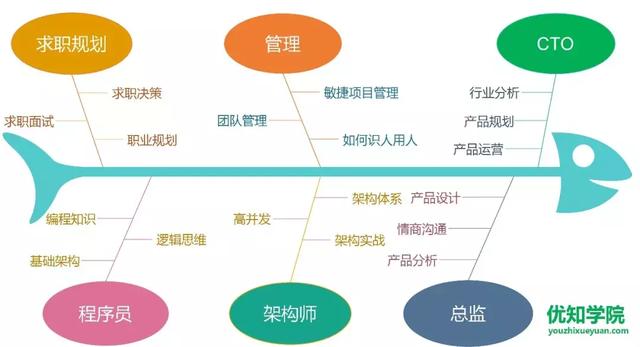
2012年5月，腾讯女性频道主编于石泓因脑溢血逝世。

**2、 如何打破35岁魔咒？**

古人云：生于忧患，死于安乐。我从事软件开发行业已经近13年，从一线大厂的程序员、高级研发经理、架构师直至CTO，即便一路走来顺风顺水，但职业发展的压力从未远离过我，生怕自己的能力结构跟不上这个行业和时代节奏，从不敢懈怠。

如果在事业上追求安乐和稳定，很可能在未来会迎来巨大的危机。商业是不相信“眼泪”的，企业在面对生存和发展危机时，没有任何温情可言。对于无法给公司创造利润的人，无论是25岁，还是35岁，都会被淘汰。我认为，35岁是否魔咒，因人而异，如果你谋定而后动、提前布局，35岁反而是一个硕果累累的年龄，甚至有可能提前实现财富自由。

程序员常见发展路径有两种：管理路径、技术路径，以下，enjoy:



**3、 关于投资理财**

很多人认为投资理财需要很多钱，这是不正确的，会理财的人，小钱可以积累到大，不会理财的人, 大钱也会消耗到光。

投资理财第一条是风险管理, 规避风险永远放在第一位，没有风险管理就像在刀口舔血一样，深入研究相关行业，选择绩优、低估值、具未来发展潜力的产品，这个产品可以是股票，可以是房产，也可以是投资入股一些小微创业型公司。

再者，投资理财要量力而为, 不要做超过你能力所能负荷的事情,譬如很多人几年前玩配资，我一个朋友在2015年买股票，本金200万，通过配资快速赚了500万，最终结果却是盈利、本金全部亏完，另外还向亲戚借了100万欠款（真实故事），我给大家的建议是不跟风、不盲从、不浮躁，多关注经济,别错把投机当投资, 这样还不如定存来得可靠安全。

**4、 生命中不仅只有代码**

人生道路上，乐观和焦虑这两种看似矛盾的情绪时常相伴，有时候乐观多一些，有时候焦虑多一些，焦虑的不是生存而是发展。面对焦虑，我的方法是学习、运动。

学习是消除内心焦虑的良方，比如学习新的技术、学习投资理财、学习哲学、阅读历史，增加自己的知识广度，人生不是只有一条路, 你得为自己想好方方面面, 而坚持学习能帮助你跳脱现况。

跑步时分泌的多巴胺，能够给人带来积极、快乐的情绪，始终让人保持一个比较良好的生活及工作状态，近两年来，我几乎坚持每天晨跑，对此深有体会。

我们应该主动去拓展生命的维度和广度，我相信，一个始终保持学习的人，一个坚持锻炼身体的人，是不会被生活“清理”、不会被时代淘汰的，我有这样的信心，希望你也有。

**5、 丢掉伪中产阶级的包袱**

众所周知，程序员的收入高，毫不客气地说，这个行业一直自带光环，但这个光环距离我们实现财富自由、时间自由还很远，程序员的固定工资收入不足以支撑我们过上中产阶级的生活，客观审视这个事实，摆正心态，正确对待。

人到中年，对于多数人来说，增加非工资性收入是给自己在船沉之前准备好的救生衣。非工资性收入，顾名思义就是不依赖于你的本职工作而获得的收益。比如：非工资性收入能够覆盖家庭开支，非工资性收入超过个人的工资性收入，非工资性收入超过全家的工资性收入等。因此，抓住每一个机会增加非工资性收入是人到中年不得不考虑的问题。

**6、 结束语**

生命可贵，压力再大，生活再难，活着即希望。

## **技术人成长的三大原则**

我自己是工程师出身，一路就像游戏打怪一样，不断去突破自己的职能舒适区，技术这条路我用了整整10年去把它全部打通，这些经验看似平淡，作为一个过来人，在你每一次新的选择的时候，你就会发现它的价值所在。

近期，我收到各种关于职场的问题:

大学毕业IT哪类职位更有前途？

从IT培训学校出来，基本等于失业？

工作3年了想做架构师，怎么学？

工作了5年，想转型管理很难取舍？

工作几年了，想转型产品？

工作几年了，想转型运营？

作为一个过来人，我结合自己10年的工作经验，谈谈如何去做出选择和取舍，真正有智慧的人，都是做减法。

**先从起步谈起：**

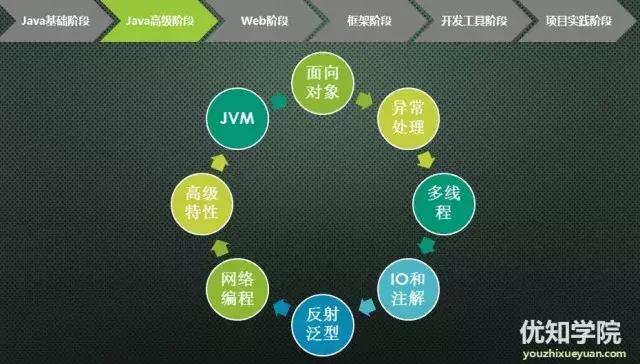


**23-27岁：技术积累阶段(初级到高级工程师)**

假设本科23岁毕业，那么工作的前5年对你来说就是打基础的阶段。在这5年时间里面，你要积累足够的技术底子，打磨自己的技术实力，成为某一个技术细分领域的牛人。

**技术基础相关的掌握牢固**

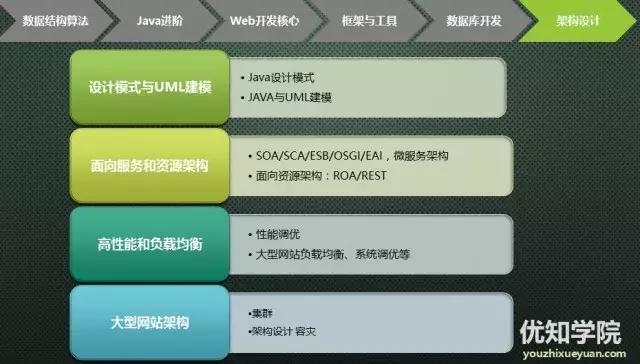
例如，很多技术语言基础和高级知识，算法、常用工具、框架原理、代码运行周期、内存回收机制等等，类似这些基础知识。



**27-30岁：形成思维方法论和知识体系的阶段**

当你积累足够的技术实力，例如超过10万行代码以后，你应该形成了自己的思维方法论和自己独立的学习技巧，任何新的技术在你眼中都能迅速的看到技术的本质，快速吸收成为你的知识体系的一部分。

举一个思维方法论例子，比如给你一个1000万的pv访问的网站架构，你是否具备这样的架构设计能力，这就会涉及到一套相关的架构设计方法，大体应该是怎样的：



**架构设计思维方法:**

* 数据库拆分原则
* 工程拆分原则
* 性能调优原则
* 性能评估原则
* 容错原则
* 框架选择原则
* 安全设计原则
* 技术方案等等

**业务分析思维方法:**

* 各种业务流程图
* 各种产品模块
* 各种你的理解

**30岁-35岁：转型阶段**

这个阶段需要突破自己，转型管理，或者深入技术架构师(深入业务)、转型[技术总监](http://youzhixueyuan.com/tag/%e6%8a%80%e6%9c%af%e6%80%bb%e7%9b%91)、产品总监、CTO等。



30岁-35岁是每一个技术人最宝贵的阶段，逐渐的不再满足于纯技术领域的探索，而是开始探索技术背后的事情：什么技术在什么样的场合能够发挥最大的价值；技术团队应该怎样构建；技术在公司未来的价值？创业公司对技术的需求？支撑技术的产品，怎样学习做产品？支撑产品背后的运营体系是怎样建立的？商业社会对技术、产品、运营等的是怎样的认知？

也有很多朋友在这个阶段，选择自己的舒适区，不敢尝试，不愿意迈出自己的舒适区，时间一晃就过去了，基本没有什么新的横向扩展积累，越往后走路约窄，其实这就是典型的身体转型，思维其实根本没有跟上。

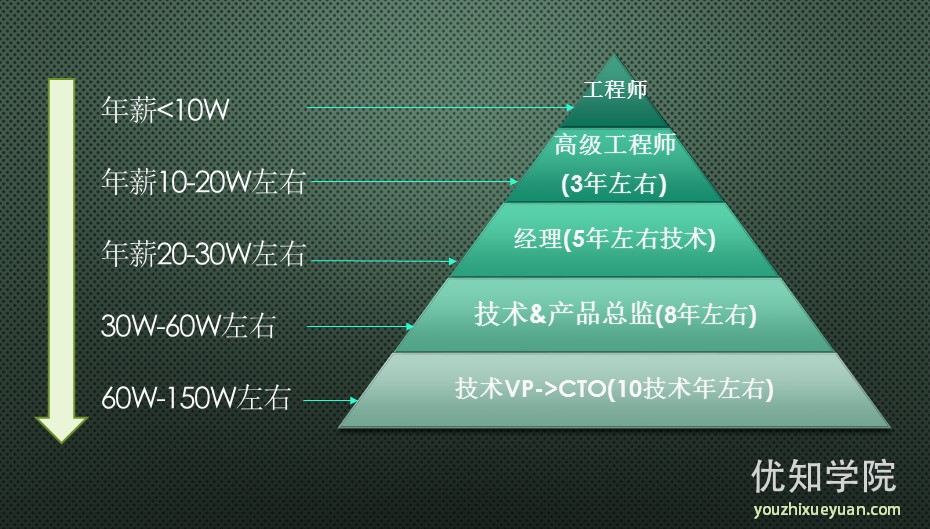
以上每一个阶段，对应技术人都特别重要，早期把技术做扎实，在中期积累一套自己的思维方法，在30岁之后，需要勇气去突破自己的舒适区，扩展自己的核心能力圈。

## **技术人的成功之路**

同理心，我自己是技术出身，也希望未来有更大的发展，当然，我相信你也想通过技术这个出身，去找寻自己的未来。

以下是技术的正常路径，你可以结合自己的能力和薪资结构，看看自己己处于第几个阶段，如果你的工作年限与以下薪资没有很好的匹配，我想也许你应该好好想想自己的问题出在哪里？

是能力的问题，还是没有合适的机会展示自己的能力，还是自己没有稳扎稳打，没有打下坚实的基础就开始梦想高薪，还是自己缺乏坚持和定力。



技术的正常路径，我其实写得非常多了，基本很清晰了，我们也可以思维发散下，以下是非典型技术出身的朋友发展路径，你也可以结合看看自己处于哪类：

1.程序员–高级程序员–架构师

2.程序员–开发组长–技术经理

3.程序员–项目经理 –技术总监

4.程序员–项目经理–敏捷咨询师

5.程序员–产品经理–产品总监

6.程序员–运营人员–运营经理

7.程序员-运营经理–运营总监

8.程序员-设计师-设计总监

9.程序员-市场人员-商务总监

我个人走的路是技术，然后结合管理和产品发展，基本也算有意识的这样去打造自己的未来，转型管理算是一个比较大的转变，但并不算最大的，最大的转变是在产品上，这一部分我以后会分享一部分(主要还是大家还没有遇见这样的问题，等你到了这个阶段自然会去深度思考)。

我自己带队也好，自己发展也好，其实都是本着技术服务产品，产品服务用户的理念，并且始终坚定的这样去推进。

其实，很多人都知道这个道理，但是就是很难把思路调整过来，毕竟拿着IT的高工资，你说我有必要去扩展产品运营这块的能力圈吗？

或许你会说未必，从大的方面看，大公司躺着赚钱的时候是最缺乏创新的时候，缺乏动力去提升产品，其实小到个人同样如此，当你很舒适的时候，其实是你最脆弱的时候，最缺乏动力的时候，往往你的未来之路就将遇见很大的阻碍，而这个阻碍你往往根本不清楚，也根本不愿意去思考，因为你完全沉浸在目前的喜悦中。

产品只是一个思路而已，主要还是扩大自己的视野，从而更清楚的看待自己的一切，理性去看待这一切，你离真相才会越近。

追求真相才是我们不断进步的源动力，不管是你工作还是生活。

人性是懒惰的，是不愿意去迎接挑战的，是不愿意逆水行走的，是不愿意克服自己的思维惰性的。

如果你能早一点认识到这一点，你就知道所谓的成功学，根本就是扯淡，真实的世界，都是需要大量的坚持、大量的学习、大量的实践。。。是一个从渐悟到顿悟的过程，如果你现在还没有顿悟这个领悟能力，那就抓紧发展自己，不断修炼自己，耐心去等待顿悟的时刻。

总之，美好的事物是需要付出极大的努力和经历极大的挫败感，然后再继续前行，继续提高自己，然后百折不挠的去追寻，这才是可实施的未来。

# ****技术人的未来****

我发现在工作中，不管是工作超过1年的还是3年5年，甚至8年的技术出身的朋友，依然从不看重职业规划的重要性，依然我行我素，不去遵循发展的规律，也有很多朋友就是为了跳槽而跳槽，从没有目标性，是时候警醒自己了，重视自己的优势和劣势，为自己下一步发展提前准备，这样的人生才有未来！

## 首先，认清外在误区

**第一大误区:技术人是吃青春饭**

这是一个误区，在中国有很多人都认为IT行业是吃青春饭的，如果过了30岁就很难有机会再发展下去！这其实是一个认识的误区，其实现实并不是这样。你完全可以继续进阶，往技术专家发展、还可以往产品、管理、运营、行业专家等方向。

**第二大误区：越老越吃香**

犹如老中医，越老越吃香，这是另一端误区。

越老并不代表越有思想，如果你没有汲取，没有不断反思，不断挑战自我的勇气，也许你有的只是不断重复的工作经验，仅此而已。

例如之前的华为计划裁掉40岁以上的程序员，华为肯定不会裁掉所有40岁以上的技术人员，但是，从市场的角度，可以算是给30岁左右的技术人敲响了警钟，未来，你是属于哪一类人群，的确需要当下认真思考你的职场核心竞争力。

**第三大误区：做技术就应该专注于技术**

这更是一大误区，不管你是否往技术专家发展，也更需要懂得业务，更需要懂得技术支撑的产品逻辑，不管是现在还是未来，懂得业务和产品，都将是你在公司或者市场的核心竞争力。

**首先，需要认清技术的价值。**

技术的存在是为了支撑产品，产品和运营是互相支撑，最终都是为了服务于用户。

技术始终是服务于用户的一个手段，而不是所有。你只有认清自己工作的价值，才能摆正心态。

例如，公司内部存在大量部门协调的事情，技术、产品、运营、市场，大家彼此之间是相辅相成的关系，而不是对立的关系。认识到这一点，对你的职场好处远大于坏处。

## 其次，遵循发展周期

对于技术人员来说，也需要认清技术的岗位发展方向和周期，如果经常跳周期发展，将适得其反。

一般一名IT人员的发展轨迹如下：

**第一阶段:打基础**

22-25岁,深入研究技术,熟练掌握一们以上语言以及相关工具，能很好的完成基本工作。

**第二阶段：扩大纵向能力**

25-30岁,成为项目小组技术带头人，至少是一名优秀的工程师 。

对技术发展方向已经很清楚，自身的方向有一定的认知。

**第三阶段：横向和纵向互相加深**

30-35岁，通常职业发展会有变化:

1. 成为一名架构师，技术专家线路；
2. 成为项目经理,逐步学会管理队伍；
3. 也可能成为技术型销售人员(售前和售后)
4. 往管理方向：技术经理、总监、 CTO；
5. 往产品方向：资深产品经理、产品总监。
6. 往运营和市场方向。

以上职位，大部分存在相互纵深发展。

例如：架构师往往也是产品、行业专家，对产品的把控和行业的判断往往不一定比产品经理差，甚至更强；技术经理，也具备很强的项目协调能力和项目管理能力。

## 认清自己的优势

人生如滚雪球，重要的是找到很湿的雪和很长的山坡，这是巴菲特的名言。其实，不仅仅只适合于投资，也适合于技术人的未来。

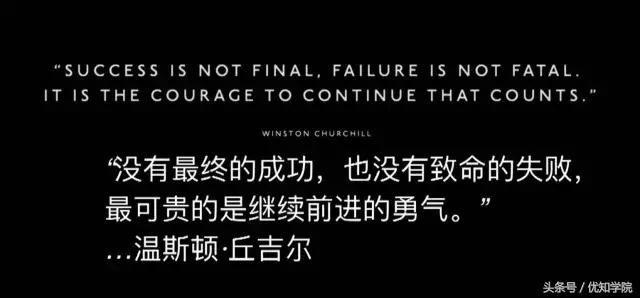
每个人都需要认清自己的立足点，在自己不同的发展时期，配合公司的发展时期，找属于你的有立足点。只有站稳了这个立足点，你才可以更好的开始延伸自己别的领域。

在当下，结合如今的发展趋势，我提供一个当下的思考，**技术人未来的核心立足点将来自于跨界的竞争：也许这才是30岁以后你的核心竞争力。**比如：你拥有雄厚的行业洞察力，敢于行业跨界，本身行业的人士往往无法与你竞争抗衡，跨界不是你的劣势，反而才是你巨大的优势，**这个时候我们竞争的不仅仅是一个维度，而是同时多个维度的PK。**

## 站在巨人的肩膀上，继续前行

站在巨人的肩膀上，你会看的更高看的更远，从而避免很多弯路，弯路过多那是对时间的浪费。

不断从你身边的牛人身上汲取过往的经验和教训，找到一个你可以参考的榜样继续前行。



**继续前行，不断去挑战自己，超越自己，这才是我们的未来！**

## **程序员真的只能干到35岁？——我的35岁危机度过之道！**

在我还是25岁的时候，我和很多程序员一样，其实都有同样的担心，老了以后怎么办？写代码难道还能写到35岁？

难道我这一辈子就做一个程序员就完了吗？难道这条路就不能走出更多的张小龙(foxmail开发者)、周鸿祎(最早期的程序员)、行癫(淘宝、天猫等大boss，淘宝第一代程序员)…?

难道我就不能走出一条属于自己印记的路？

不要在你老去的那一天后悔你居然对生活不曾有奢望，更不曾为此努力过。

让自己变得更加增值，这才是王道！

### 危机的根源

我一直相信这样一句话：**真正的危机，来源于在正确的时间做不正确的事。没有在正确的时间，为下一步做出积累，这才是危机的根源。**

比如，当你迈过了30岁这个坎，你的能力还局限于程序的范围，也没有积累好自己的深度以及视野。如果真是这样，我想我一定会每天夜不能寐，每天唾骂自己，你特么究竟每天在干神马。

我也一直相信这样一句话，**当你把你的眼泪流干了，你唯一剩下的就只有行动，只有行动和努力了**。

如果你正在这条成长路上的朋友，晚醒不如早醒，这就是我想说的。千万别等到中年才发现自己没有建立好自己的护城河，这个时候才知道努力。**在自己努力的阶段，不仅不努力反了选择了纵容自己，这才是危机的根源。**

### 做正确的事情

我就是一个土生土长的程序员，最早做的前段开发(第一代css、js程序员),然后开始转做后端开发(.net、php、java)，又转到架构师以及管理，然后再做产品…以下就是我从有限的10年左右提炼出的部分建议。这些也许对你来说不一定有用，但一定是发自我内心的建议。

### 23-27岁：技术积累阶段

假设本科23岁毕业，那么工作的前5年对你来说就是打基础的阶段。在这5年时间里面，你要积累足够的技术底子，打磨自己的技术实力，**成为某一个技术细分领域的牛人**。

我自己依然记得当年这样的场景：蹲在出租房里面每晚每晚的啃技术文档，在公司连续打地铺干通宵的场景。

只有真正经历过贫穷，经历过苦难的人，才会越早就懂得努力才自己这一生唯一的出路和唯一的机会。忘了谁说的了，大概就是这个理。

### 27-30岁：形成思维方法论和知识体系的阶段

当你积累足够的技术实力，例如超过10万行代码以后，你应该形成了自己的思维方法论和自己独立的学习技巧，任何新的技术在你眼中都能迅速的看到技术的本质，快速吸收成为你的知识体系的一部分。

你只要前一个阶段打的基础足够牢固，这个阶段你一定会开始蚕食别的领域，这是一定的。逐渐的不再满足于纯技术领域的探索，而是开始探索技术背后的事情：什么技术在什么样的场合能够发挥最大的价值；技术团队应该怎样构建；技术在公司未来的价值？创业公司对技术的需求？支撑技术的产品，怎样学习做产品？支撑产品背后的运营体系是怎样建立的？商业社会对技术、产品、运营等的是怎样的认知，商业社会有什么的特定规律?也许还有更多…

在这个阶段，积累技术对你来说简直是小菜一碟，**你更需要磨练的是思考能力**，**形成自己的思维方法和知识体系**，这将是你帮助你一生的武器。

**30-33岁这个阶段，最关键的是了解自己**。知道自己擅长什么，自己的优势在哪里，**怎样才能够最大化自己的价值**，**同时也知道自己有什么重大缺陷。未来是扬长避短还是不断避免自己的缺陷，这些需要提前想清楚。**一旦想清楚，想透了，就剩下行动。你可以选择避开做自己不擅长的事情，不勉强自己，不让自己处于对自己不利的环境。当然，你也可以选择不断补短，把自己的不擅长点变成自己最擅长的。这个需要结合自己的情况来慎重选择，最终看你要走什么样的一条路，是打算做技术专家、管理者、还是技术结合产品，还是创业者。

### 走自己的路

程序员也好，设计师也好，架构师也好，产品经理也好，这都是一份职业。职业一定有他的职业基础操守，这个是需要遵守的。千万别程序员做不到就开始换做产品，换做运营，这样的思维不管你做什么工作，也许你什么都做不好。

所以，没有做好本职工作之前，别轻易谈转行。

也可以好好琢磨琢磨，很多程序员出身的，哪个是程序员都做不好，就敢说自己产品很牛逼，我就是一个运营天才。

从程序员开始，我看挺好的，干不动还可以干程序员么。

最后，再送给大家一个我的一个锦囊。

每天坚持锻炼，比如：跑步、游泳等，至少一项运用可以坚持3个月以上。

你会逐渐发现，也许这些都不是个事。

## 程序员怎么升职加薪？

### 程序员升职加薪之路

首先我们一起看看薪资结构升职，这也是大家最关心的话题。

以上的薪资结构，可以算是一份参考，还要根据不同的地区(北上广深和内陆)，以及公司(BAT TMD 独角兽 创业公司等)再作微调。

刚毕业的1年左右的程序员朋友，基本都在10W年薪左右，这个阶段都在忙于生存压力，能把公司的活干好就不错了，干好项目不要出大篓子。

再工作2年后，3年左右，对程序员这份工作有更多的了解，不再局限于生存，开始有自己的想法了，编程代码和算法功底也在加强，这个阶段经常重构自己写的代码算是常态。我也一样，在这个阶段经常看见之前写的代码就想重构，偶了用点新学会的设计模式以及高效率算法来提高重用性。

在如今万众创业这个阶段，工作3年左右的程序员拿一份20W左右的年薪很正常，在30W也不少，特别是前段时间APP的开发，ios 工作3年左右还不错的，真是抢手货啊，我知道的很多创业公司，开价月薪2W上的很普遍。

还有，在这个万众创业的阶段，程序员跳槽太普遍了，而且频率实在是太快了，大部分人都是在跳槽涨工资，这个仁者见仁智者见智，我就不妄作评论了。

### 技术线路升职之路

再工作2年，到5年这个坎了，基本后面的发展也比较清晰了，往后升职的空间也就这几个方向。

往架构师发展，技术纵深，成为某一领域的技术专家，薪资么往60W进军。当然，这肯定需要看公司的需求了，大部分的创业公司，肯定给不了你这个价，这需要看缘分以及你的实力和运气。

还有一部分也很明确，想往管理发展，这里涉及的职位相对比较多，项目经理、技术主管、[技术经理](http://youzhixueyuan.com/tag/%e6%8a%80%e6%9c%af%e7%bb%8f%e7%90%86)、[技术总监](http://youzhixueyuan.com/tag/%e6%8a%80%e6%9c%af%e6%80%bb%e7%9b%91)等，[技术经理](http://youzhixueyuan.com/tag/%e6%8a%80%e6%9c%af%e7%bb%8f%e7%90%86)的缺口最大，再往[技术总监](http://youzhixueyuan.com/tag/%e6%8a%80%e6%9c%af%e6%80%bb%e7%9b%91)发展难度增大。

我知道很多做项目经理，其实不懂技术的蛮多的，很多都是从软件行业以及别的行业转行过来的，协调项目为主。当然，也有很多公司的项目经理其实是[技术经理](http://youzhixueyuan.com/tag/%e6%8a%80%e6%9c%af%e7%bb%8f%e7%90%86)，本身就懂技术而且还管理技术团队。其实，[技术经理](http://youzhixueyuan.com/tag/%e6%8a%80%e6%9c%af%e7%bb%8f%e7%90%86)比较准确，项目经理还是更多协调为主比较明确。

如果你真打算往管理方向发展，我个人比较支持走[技术经理](http://youzhixueyuan.com/tag/%e6%8a%80%e6%9c%af%e7%bb%8f%e7%90%86)这条路。所以，你至少得从程序员做起，能做到架构这个水平最好，有技术纵深，再去横向扩展你的管理水平。这样的方式，你的团队成员对你也服气，不然少不了被团队质疑，特别是空军这种情况，基本都是炮灰。

往管理方向，技术的管理相对别的管理难度在降低，如果你懂技术而且有深度，这就好办。很多不懂技术的管理者，还是避免去管理技术型的团队。这里提醒一点，如果你真往管理方向发展，我个人建议还是要彻底一点，别手抓技术不放，也对团队管理不上心。这个阶段不需要你天天冲到前线奋战，更多的是扮演教练的角色，辅助兄弟们成长，你在旁边提供协助以及后勤保障。这个道理，其实很简单，但是，做得好的不多，其实就是升级不彻底，总想着自己的技术“铁饭碗”，甚至，很多最后又回去做技术了，觉得自己不适合管理团队，喜欢纯粹的技术生活。

再工作3年左右，基本在8-10年这个坎，很多能做到[技术总监](http://youzhixueyuan.com/tag/%e6%8a%80%e6%9c%af%e6%80%bb%e7%9b%91)以及技术专家，这个是也比较普遍。这个阶段，其实也比较清楚自己的短板，比如：管理、架构、协调、情商、性格等的短板比较清楚。

### [产品运营](http://youzhixueyuan.com/tag/%e4%ba%a7%e5%93%81%e8%bf%90%e8%90%a5)转型之路

这里还有一路分支，很多转型去做产品了，往产品经理发展，如果是到了这个阶段转型产品的，基本很少有失败的，大部分都属于转型过去就不打算会回来做技术了。这里，技术转产品，有非常好的独特优势。原因很简单，在中国大部分的产品经理的工作，包括BAT TMD等一线互联网公司， 大部分都是功能性产品经理，从业务逻辑的细节再到流程等，这些细节都是程序员一行行代码自己敲出来的，比大部分产品经理还懂公司的业务。很多公司的老程序员，产品经理都要求助去他，就是因为细节在他们手里，这样的转型比较好。

其实，我个人认为这里还有一个分支，目前这个分支，这样转型的不多，但是，未来5年很有可能会变成主流也不一定。就是技术转型到[产品运营](http://youzhixueyuan.com/tag/%e4%ba%a7%e5%93%81%e8%bf%90%e8%90%a5)，请记住，是[产品运营](http://youzhixueyuan.com/tag/%e4%ba%a7%e5%93%81%e8%bf%90%e8%90%a5)，运营来驱动产品的人员。如果能转型成功，未来甭说100W年薪，200W年薪也算是基本常态。如果你对这个话题感兴趣的，欢迎到优知学院官网上留言，我们再继续探讨。

## **如何选择一家公司**

不管是刚毕业的大学生还是工作几年的职场朋友，每个人都会面临选择公司和行业的困扰，我也相信每个人都还记忆犹新你的第一份工作以及让你无比难忘的一家公司。有时候我们也盲目的所求，其实，偶尔停下来思考下你真想去的地方，再出发，也许结果将迥然不同。今天，给大家聊聊，如何选择一家公司。

### **BAT**

应届毕业生和工作3-5年阶段技术人员，优先推荐期间进入BAT较好。BAT里大家的共识是分为三类:技术优先进百度，运营进阿里，产品进腾讯，这个共识对很多人其实是没有意义的，因为你甚至一家都进不了，何谈选择。下面我谈下个人经验，也许能给你带来点思路。我是在工作第四年09年加入阿里淘宝，截止到现在，淘宝这个阶段我还是认为这个阶段为自己打下了坚实的基础和视野，成长最快的还是在这个阶段，这是我个人的一个体会。在早期阶段，你需要为自己打下坚实的基础，这个基础包含：技能、态度、沟通，还有一点至关重要，那就是“视野”，这一点的重要度，很多时候被职场的名利所忽视。视野听起来很宽泛，但如果你仔细辨认，其实一目了然，当你身边都是一流的人才的时候，你自然会学到很多，这其中的某些学到的品质，会影响你很久。  
举一个例子，淘宝的时候有一个大名鼎鼎的一位同学,我们称他为“卷爷”，花名“雷卷“”，在淘宝做技术的没有人不认识，期间做一个淘宝店铺装修的项目，与其一起做淘宝店铺装修底层架构。项目期间，卷爷特别喜欢代码review，一般我们写代码每个人每个风格，这才是真的千人千面，你很难统一，更何谈热爱代码规范。项目期间，每天卷爷都会代码review，而且标准极其严格，经常会听到他在批一个人的代码“写的给屎一样”，而且特喜欢把写代码的人叫一旁，解释一通为什么要这样写。最后，丢下一句话：代码如人，你看你穿的整整齐齐的，为什么不洗脸就出门了呢。我就坐卷爷旁边，经常听得我直冒冷汗，回头看看是否我也在犯这个错误。刚开始不理解卷爷，后来理解了，他就是好这口，因为他认为这件事对他来讲是艺术，不容别人捣乱。  
说到视野，还有一个关于子女教育很类似，为什么说女孩要富养，其实都具有异曲同工之妙，在早期你需要多带她出去见识，开阔视野，至少清楚什么是好的人和事物，这个阶段至关重要，不仅仅适合于教育也包含职场。**简要概括几点重点：**

1. 有机会快速认识很多牛人。
2. 视野开阔很多，不仅仅在技术。
3. 每次项目，你会接触潜在很多关于产品、运营、设计的知识，类似这些知识算是长期积累的过程，是经得起验证的知识体系。
4. 早期进入，你的压力一定会很大，但成长也会很快，在里面锻炼几年你会比外面工作十年的厉害很多。
5. 不管你是继续在BAT还是你准备以后离开，都是一个很好的信用背书，前提是你在里面做的还不错。
6. 任何职场都有潜规则，但这个阶段你需要做到专注，专注提升自己的实力，潜规则放在一边，这个阶段需要专注提升。
7. 如果你实在受不了太多的限制于约束，或者你完全可以独挡一片，当然你可以不进BAT，我只是告知你发展阶段的一种途径而已。

关于怎么进入BAT，我后期会简短概述一部分有效的方法。当然，你也可以走另外一条路，成长型公司。没有说哪一条是最好的，但至少结合你的当下的情况，你可以选择当下最优。

### **成长型公司**

这里我不区分是一线互联网企业还是创业型企业，统称为成长型公司。怎么区分成长型公司？一般过了A轮，初步你可以理解为成长较快。在职场早期如果你很难进入BAT，你可以考虑进入成长型锻炼。人生就是这样，很有可能，你会失之东隅收之桑榆，锻炼出来比在BAT还厉害的多，而且心智更全面，特别适合以后创业。有一种思维叫风险与收益成正比，你的风险相对于公司的风险，你的风险还是偏小。所其思考有什么风险呢，还不如安心专注提升自己。**我简要概括几件重点：**1**做好功课-选择行业**你首先需要做好功课，选择行业。不管你是互联网+(互联网改造)、互联网企业、+互联网(偏传统)，他们都对应一个行业属性。行业属性在以后比例会逐渐上升，很有可能你干着技术的活，很多人需要来请教你行业的知识。举个例子，比如你做的是旅游行业的技术，日常工作中有很多与旅游相关的产品与业务知识，除了做好你的本质工作，这些知识你也需要整理与收集。行业的知识体系你挖掘的越深，也许在一天你转岗的时候，你会感慨自己有多么的明智。2**建议选择公司-心中有顺序**优先选择互联网企业，其次互联网+(互联网链接)，再次+互联网，再次传统企业和国企。逻辑很简单,如果你是做技术的，你可以看见哪家公司，技术是核心部门。没有哪家公司不对自己的核心部门长期投入的事。技术的利用在互联网企业里，用户基数更大，这个时候技术才能发挥作用。3**认清优势-不断积累**因为除了做技术，你很有可能还需要兼职产品，甚至运营的工作。如果你某一天要走上创业这条道路，那将帮助你不少。如果你只是通过该公司成功踏上类似BAT这条船，也不错。希望今天的内容能对你带来点思考。总之，毕业选择加入一家公司，首先的有自己的一个基本判断，进入什么样的行业以及对应的职位，首先需要过滤清楚，然后再行动。

## Java程序员的发展之路和职业规划

在互联网做技术的朋友，往往没有足够的重视，职业规划其实一点都不虚，而是一件非常实在的事情，如果你不是每次碰墙再反思职业规划，而是提前3年左右作出下一步的规划，你早已经走出了一条属于自己的路。 以下是我自己的一部分思考。

### **技术人的困扰**

技术人的成长之路并非一帆风顺，如果你在成长过程中经常会问自己诸如此类的的问题，我想你应该停下片刻认真思考一下你的职业规划，再重新出发。

1. 发展前途比较迷茫，是继续做好技术，还是准备转岗，或者换一个行业。
2. 如何选择适合自己的岗位。
3. 跳槽完了问题还是没解决。
4. 想换岗，却不知道从何谈起。
5. 甚至想换一个行业，来一次自我救赎。
6. 工作很烦心，工作已经毫无激情可言。
7. 一边安于现状，一边抱怨机会 太少。
8. 不安于现状，希望赶快脱贫致富。
9. 想换岗，怎么换，怎么行动。

如果你在以上问题中占据一项或者多项，你已经碰到了职业规划这堵墙，怎么应对，怎么思考，怎么解决，希望此文能给大家带来一点反思。

### **认清外在误区**

**第一大误区:技术人是吃青春饭**这是一端误区。在中国有很多人都认为IT行业是吃青春饭的，如果过了30岁就很难有机会再发展下去！这其实是一个认识的误区，其实现实并不是这样。你完全可以继续进阶，往技术专家发展、还可以往产品、管理、运营、行业专家等方向。

**第二大误区：越老越吃香**犹如老中医，越老越吃香，这是另一端误区。越老并不代表越有思想，如果你没有汲取，没有不断反思，不断挑战自我的勇气，也许你有的只是不断重复的工作经验，仅此而已。例如最近传言华为计划裁掉40岁以上的程序员。华为肯定不会裁掉所有40岁以上的技术人员，但是，从市场的角度，可以算是给30岁左右的技术人敲响了警钟，未来，你是属于哪一类人群，的确需要当下认真思考你的职场核心竞争力。

**第三大误区：做技术就应该专注于技术**这更是一大误区，不管你是否往技术专家发展，也更需要懂得业务，更需要懂得技术支撑的产品逻辑，不管是现在还是未来，懂得业务和产品，都将是你在公司或者市场的核心竞争力。首先，需要认清技术的价值，技术的存在是为了支撑产品，产品和运营是互相支撑，最终都是为了服务于用户。技术始终是服务于用户的一个手段，而不是所有。你只有认清自己工作的价值，才能摆正心态。例如，公司内部存在大量部门协调的事情，技术、产品、运营、市场，大家彼此之间是相辅相成的关系，而不是对立的关系。认识到这一点，对你的职场好处远大于坏处。再举一个例子，现在不管是BAT还是创业公司，都在大力提倡敏捷开发，不管是小步快跑还是快速迭代，其实提倡敏捷开发的背后的的核心逻辑：是给大家创造一个懂得团队协作的环境，否则天天站会做什么呢，你以为只是提早发现进展风险和BUG？

### **遵循发展周期**

对于技术人员来说，也需要认清技术的岗位发展方向和周期，如果经常跳周期发展，将适得其反。一般一名IT人员的发展轨迹如下：

**第一阶段:打基础**22-25岁,深入研究技术,熟练掌握一们以上语言以及相关工具，能很好的完成基本工作。

**第二阶段：扩大纵向能力**25-30岁,成为项目小组技术带头人，至少是一名优秀的工程师 。对技术发展方向已经很清楚，自身的方向有一定的认知。

**第三阶段：横向和纵向互相加深**30-35岁，通常职业发展会有变化:

1. 成为一名架构师，技术专家线路；
2. 成为项目经理,逐步学会管理队伍；
3. 也可能成为技术型销售人员(售前和售后)
4. 往管理方向：技术经理、总监、 CTO；
5. 往产品方向：资深产品经理、产品总监。
6. 往运营和市场方向。

以上职位，大部分存在相互纵深发展。例如：架构师往往也是产品、行业专家，对产品的把控和行业的判断往往不一定比产品经理差，甚至更强；技术经理，也具备很强的项目协调能力和项目管理能力。

### **找到自己的方向**

人生如滚雪球，重要的是找到很湿的雪和很长的山坡，这是巴菲特的名言。其实，不仅仅只适合于投资，也适合于技术人的职业规划。每个人都需要认清自己的立足点，在自己不同的发展时期，配合公司的发展时期，找属于你的有立足点。只有站稳了这个立足点，你才可以更好的开始延伸自己别的领域。在当下，结合如今的发展趋势，我提供一个当下的思考，技术人未来的核心立足点将来自于跨界的竞争，跨界不仅仅包含行业跨界，也包括职位的跨界，以及更多视野的跨界。

### **站在巨人的肩膀上**

站在巨人的肩膀上，你会看的更高看的更远，从而避免很多弯路，弯路过多那是对时间的浪费。也许当下，站在巨人的肩膀上，这将是最好的职业规划解决方案。不断从你身边的牛人身上汲取过往的经验和教训，找到一个你可以参考的榜样开始你的职业生涯。

# ****数据库****

## 数据库主从同步的3种一致性方案实现，及优劣比较

### 数据主从同步的由来

互联网的很多业务，特别是在高并发的场景下，基本都是读远远大于写，如果数据库读和写的压力都同在一台主机上，这显然不太合理。

于是，把一台数据库主机分为单独的一台写主库（主要负责写操作），而把读的数据库压力分配给读的从库，而且读从库可以变为多台，这就是读写分离的典型场景如下：



为了进一步的降低数据库端的压力(高并发的瓶颈)，这个时候也会在业务层部署分布式缓存集群(redis、memcached)等，把读的压力转移给应用服务器端，其实与数据主从的设计是遵循同一个原则，降低后端数据库的压力。

**问题：**

读写分离提高了资源的利用效率的同时也引出了一个问题，就是由于延时（网络传输，操作）而引起的数据库主从不一致的问题，以下会详细谈相关的数据一致性解决方案。

### 数据同步一致性解决方案

**1.半同步复制**

办法就是等主从同步完成之后，等主库上的写请求再返回，这就是常说的“半同步复制”。

**实现方案**

mysql的半同步复制方案，下面我以mysql为例介绍。



**MySQL半同步复制**

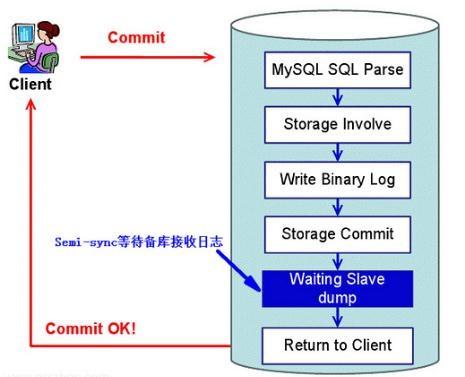
MySQL的Replication默认是一个异步复制的过程，从MySQL5.5开始，MySQL以插件的形式支持半同步复制，我先谈下异步复制，这样可以更好的理解半同步复制。

**1）异步复制**

MySQL默认的复制是异步的，主库在执行完客户端提交的事务后会立即将结果返给给客户端，并不关心从库是否已经接收并处理，这样就会有一个问题，主如果crash掉了，此时主上已经提交的事务可能并没有传到从库上。

**2）半同步复制**

介于异步复制和全同步复制之间，主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待至少一个从库接收到并写到relay  
log中才返回给客户端。相对于异步复制，半同步复制提高了数据的安全性，同时它也造成了一定程度的延迟，这个延迟最少是一个TCP/IP往返的时间。所以，半同步复制最好在低延时的网络中使用。



**半同步复制原理：**

* 事务在主库写完binlog后需要从库返回一个已接受，才放回给客户端
* mysql5.5版本以后，以插件的形式存在，需要单独安装
* 确保事务提交后binlog**至少传输到一个从库**
* 不保证从库应用完成这个事务的binlog
* 性能有一定的降低
* 网络异常或从库宕机，卡主库，直到超时或从库恢复

**该方案优点：**

利用数据库原生功能，比较简单

**该方案缺点：**

主库的写请求时延会增长，吞吐量会降低

**2.数据库中间件**



**流程：**

1）所有的读写都走数据库中间件，通常情况下，写请求路由到主库，读请求路由到从库

2）记录所有路由到写库的key，在主从同步时间窗口内（假设是500ms），如果有读请求访问中间件，此时有可能从库还是旧数据，就把这个key上的读请求路由到主库。

3）在主从同步时间过完后，对应key的读请求继续路由到从库。

**相关的中间件有：**

1）canal:是阿里巴巴旗下的一款开源项目，纯Java开发,基于数据库增量日志解析，提供增量数据订阅&消费，目前主要支持了MySQL。

2）otter：也是阿里开源的一个分布式数据库同步系统，尤其是在跨机房数据库同步方面，有很强大的功能。它是基于数据库增量日志解析，实时将数据同步到本机房或跨机房的mysql/oracle数据库。

**两者的区别在于：**

otter目前嵌入式依赖canal，部署为同一个jvm，目前设计为不产生Relay Log。

otter目前允许自定义同步逻辑，解决各类需求。

**该方案优点**

能保证绝对一致

**该方案缺点：**

数据库中间件的成本较高

### **缓存记录写key法**



**写流程：**

1）如果key要发生写操作，记录在cache里，并设置“经验主从同步时间”的cache超时时间，例如500ms

2）然后修改主数据库

**读流程：**

1）先到缓存里查看，对应key有没有相关数据

2）有相关数据，说明缓存命中，这个key刚发生过写操作，此时需要将请求路由到主库读最新的数据。

3）如果缓存没有命中，说明这个key上近期没有发生过写操作，此时将请求路由到从库，继续读写分离。

**该方案优点：**

相对数据库中间件，成本较低

**该方案缺点：**

为了保证“一致性”，引入了一个cache组件，并且读写数据库时都多了缓存操作。

以上就是数据库主从同步一致性方案详解，更多分布式大数据分布式文件系统和分布式数据库的一致性可以参考：分布式数据库数据一致性的原理、与技术实现方案

## NoSQL和SQL的区别，NoSQL的使用场景和选型比较

### 什么是NoSQL

NoSQL，指的是非关系型的数据库。NoSQL有时也称作Not Only SQL的缩写，是对不同于传统的关系型数据库的数据库管理系统的统称,它具有非关系型、分布式、不提供ACID的数据库设计模式等特征。

NoSQL用于超大规模数据的存储。（例如谷歌或Facebook每天为他们的用户收集万亿比特的数据）。这些类型的数据存储不需要固定的模式，无需多余操作就可以横向扩展。

### SQL 和 NoSQL 的区别



SQL数据库适合那些需求确定和对数据完整性要去严格的项目。NoSQL数据库适用于那些对速度和可扩展性比较看重的那些不相关的，不确定和不断发展的需求。简单来说就是：

* **SQL是精确的。**它最适合于具有精确标准的定义明确的项目。典型的使用场景是在线商店和银行系统。
* **NoSQL是多变的。**它最适合于具有不确定需求的数据。典型的使用场景是社交网络，客户管理和网络分析系统。

### SQL和Nosql的选型和比较

**1.关系型数据库和非关系型数据库**

SQL (Structured Query Language) 数据库，指关系型数据库。主要代表：SQL Server，Oracle，MySQL等。

NoSQL（Not Only SQL）泛指非关系型数据库，主要代表：MongoDB，Redis等。

**2.关系型数据库适合存储结构化数据**

如用户的帐号、地址等：

1）这些数据通常需要做结构化查询，比如join，这时候，关系型数据库就要胜出一筹

2）这些数据的规模、增长的速度通常是可以预期的

3）保证数据的事务性、一致性要求。

**3.NoSQL适合存储非结构化数据**

如发微博、文章、评论：

1）这些数据通常用于模糊处理，如全文搜索、机器学习

2）这些数据是海量的，而且增长的速度是难以预期的，

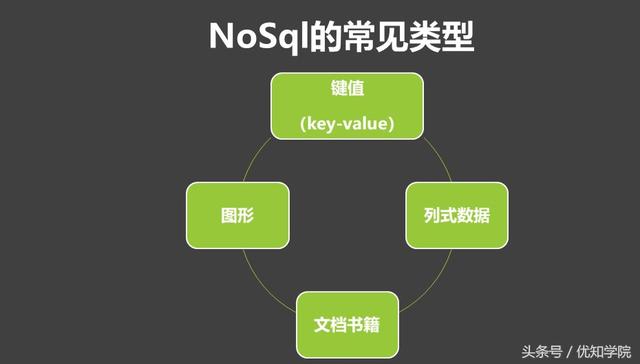
3）根据数据的特点，NoSQL数据库通常具有无限（至少接近）伸缩性

4）按key获取数据效率很高，但是对join或其他结构化查询的支持就比较差

目前许多大型互联网项目都会选用MySQL（或任何关系型数据库） + NoSQL的组合方案。

### NoSQL的常见类型和比较

有四种常见的 NoSQL 数据库类型：列式、文档、图形和内存键值。



**1.列式数据**

顾名思义，是按列存储数据的。最大的特点是方便存储结构化和半结构化数据，方便做数据压缩，对针对某一列或者某几列的查询有非常大的IO优势。

1）对应的nosql： HBase,BigTable等。

2）典型应用场景：按列存储，针对某一列或者某几列的查询有非常大的IO优势。

3）优点：查找速度快，可扩展性强，更容易进行分布式扩展。

4）缺点：功能相对局限。

**2.文档数据库**

旨在将半结构化数据存储为文档，通常采用 JSON 或 XML 格式。与传统关系数据库不同的是，每个 NoSQL 文档的架构是不同的，可让您更加灵活地整理和存储应用程序数据并减少可选值所需的存储。

1）对应的nosql：CouchDB, MongoDb

2）典型应用场景：存储类似JSON格式的内容，可对某些字段建立索引功能，是最像关系型的数据库。

3）优点：数据结构要求不严格，表结构可变，不需要像关系型数据库一样需要预先定义表结构。

4）缺点：查询性能不高，而且缺乏统一的查询语法。

**3.图形数据库**

可存储顶点以及称为边缘的直接链路。图形数据库可以在 SQL 和 NoSQL 数据库上构建。顶点和边缘可以拥有各自的相关属性。

1）数据模型：图结构

2）典型应用场景：社交网络，推荐系统等。专注于构建关系图谱，善于处理大量复杂、互连接、低结构化的数据，数据往往变化迅速，且查询频繁。

3）优点：利用图结构相关算法。比如最短路径寻址，N度关系查找等。

4）缺点：很多时候需要对整个图做计算才能得出需要的信息，而且这种结构不太好做分布式的集群方案。

**4.内存键值存储**

可以通过key快速查询到其value。一般来说，存储不管value的格式，照单全收，是针对读取密集型应用程序工作负载（例如社交网络、游戏、媒体共享和 Q&A 门户）。内存缓存可将重要数据存储在内存中以实现低延迟访问，从而提高应用程序性能。

1）对应的nosql：Redis,Memcached等

2）典型应用场景：内容缓存，主要用于处理大量数据的高访问负载，也用于一些日志系统等等。

3）优点：查找速度快。

4）缺点：数据无结构化，通常只被当作字符串或者二进制数据。

## [MySQL慢查询优化、索引优化、以及表等优化总结](http://youzhixueyuan.com/mysql-slow-query-optimization-index-optimization.html)

### MySQL优化概述

MySQL数据库常见的两个瓶颈是：CPU和I/O的瓶颈。

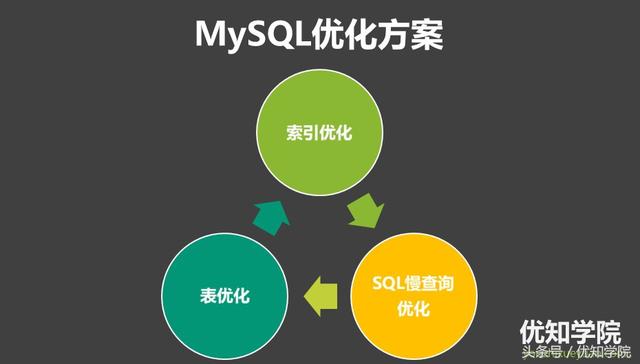
CPU在饱和的时候一般发生在**数据装入内存或从磁盘上读取数据时候**。

磁盘I/O瓶颈发生在装入数据远大于内存容量的时候，如果应用分布在网络上，**那么查询量相当大的时候那么平瓶颈就会出现在网络上。**

我们可以用mpstat, iostat, sar和vmstat来查看系统的性能状态。除了服务器硬件的性能瓶颈，对于MySQL系统本身，我们可以使用工具来优化数据库的性能。

### MySQL优化方案

**Mysql的优化，大体可以分为三部分：索引的优化，sql语句的优化，表的优化**



### 索引优化

**1.索引**

一般的应用系统，读写比例在10：1左右，而且插入操作和一般的更新操作很少出现性能问题，在生产环境中，我们遇到最多的也是最容易出现问题的，还是一些复杂的查询操作，因此对查询语句的优化是重中之重，**加速查询最好的方法就是索引。**

索引：简单的说，相当于图书的目录，可以帮助用户快速的找到需要的内容。

在MySQL中也叫做“键”，是存储引擎用于快速找到记录的一种数据结构。能够大大提高查询效率。特别是当数据量非常大，查询涉及多个表时，使用索引往往能使查询速度加快成千上万倍。

总结：索引的目的在于提高查询效率，与我们查询图书所用的目录是一个道理：先定位到章，然后定位到该章下的一个小结，然后找到页数。相似的例子还有：查字典，查地图等。

**2.索引类型**

* 普通索引

是最基本的索引，它没有任何限制。

* 唯一索引

与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。

* 组合索引

指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。

* 主键索引

是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值。一般是在建表的时候同时创建主键索引

* 全文索引

主要用来查找文本中的关键字，而不是直接与索引中的值相比较。fulltext索引跟其它索引大不相同，它更像是一个搜索引擎，而不是简单的where语句的参数匹配。fulltext索引配合match against操作使用，而不是一般的where语句加like。它可以在create table，alter table ，create index使用，不过目前只有char、varchar，text 列上可以创建全文索引。值得一提的是，在数据量较大时候，现将数据放入一个没有全局索引的表中，然后再用CREATE index创建fulltext索引，要比先为一张表建立fulltext然后再将数据写入的速度快很多。

**3.索引优化**

* 只要列中含有NULL值，就最好不要在此例设置索引，复合索引如果有NULL值，此列在使用时也不会使用索引
* 尽量使用短索引，如果可以，应该制定一个前缀长度
* 对于经常在where子句使用的列，最好设置索引，这样会加快查找速度
* 对于有多个列where或者order by子句的，应该建立复合索引
* 对于like语句，以%或者‘-’开头的不会使用索引，以%结尾会使用索引
* 尽量不要在列上进行运算（函数操作和表达式操作）
* 尽量不要使用not in和<>操作

#### SQL慢查询的优化



**1.如何捕获低效sql**

1）slow\_query\_log

这个参数设置为ON，可以捕获执行时间超过一定数值的SQL语句。

2）ong\_query\_time

当SQL语句执行时间超过此数值时，就会被记录到日志中，建议设置为1或者更短。

3）slow\_query\_log\_file

记录日志的文件名。

4）log\_queries\_not\_using\_indexes

这个参数设置为ON，可以捕获到所有未使用索引的SQL语句，尽管这个SQL语句有可能执行得挺快。

**2.慢查询优化的基本步骤**

1)先运行看看是否真的很慢，注意设置SQL\_NO\_CACHE

2）where条件单表查，锁定最小返回记录表。这句话的意思是把查询语句的where都应用到表中返回的记录数最小的表开始查起，单表每个字段分别查询，看哪个字段的区分度最高

3)explain查看执行计划，是否与1预期一致（从锁定记录较少的表开始查询）

4)order by limit 形式的sql语句让排序的表优先查

5)了解业务方使用场景

6)加索引时参照建索引的几大原则

7)观察结果，不符合预期继续从1开始分析

**2.优化原则**

* 查询时，能不要\*就不用\*，尽量写全字段名
* 大部分情况连接效率远大于子查询
* 多使用explain和profile分析查询语句
* 查看慢查询日志，找出执行时间长的sql语句优化
* 多表连接时，尽量小表驱动大表，即小表 join 大表
* 在千万级分页时使用limit
* 对于经常使用的查询，可以开启缓存

#### 数据库表优化

* 表的字段尽可能用NOT NULL
* 字段长度固定的表查询会更快
* 把数据库的大表按时间或一些标志分成小表
* 将表拆分

数据表拆分：主要就是垂直拆分和水平拆分。

水平切分:将记录散列到不同的表中，各表的结构完全相同，每次从分表中查询, 提高效率。

垂直切分:将表中大字段单独拆分到另外一张表, 形成一对一的关系。

#### 总之：

Mysql的优化主要就在于：索引的优化，sql语句的优化，表的优化，在高并发网络环境下，除了优化数据库外，还会涉及到分布式缓存，CDN，数据库读写分离等高并发优化技术。

## MySQL有哪些存储引擎，各自的优缺点，应用场景

经常面试都会问到MYSQL有哪些存储引擎，以及各自的优缺点。今天主要分享常见的存储引擎：MyISAM、InnoDB、MERGE、MEMORY(HEAP)、BDB(BerkeleyDB)等，以及最常用的MyISAM与InnoDB两个引擎 ，文章尾部有两者的详细比较。

### MySQL常用存储引擎介绍

**1.InnoDB 引擎(MySQL5.5以后默认使用)**

MySQL 5.5 及以后版本中的默认存储引擎，他的优点如下：

* 灾难恢复性好
* 支持事务
* 使用行级锁
* 支持外键关联
* 支持热备份
* 对于InnoDB引擎中的表，其数据的物理组织形式是簇表（Cluster Table），主键索引和数据是在一起的，数据按主键的顺序物理分布
* 实现了缓冲管理，不仅能缓冲索引也能缓冲数据，并且会自动创建散列索引以加快数据的获取
* 支持热备份

**2.MyISAM引擎**

**特性如下：**

* 不支持事务
* 使用表级锁，并发性差
* 主机宕机后，MyISAM表易损坏，灾难恢复性不佳
* 可以配合锁，实现操作系统下的复制备份、迁移
* 只缓存索引，数据的缓存是利用操作系统缓冲区来实现的。可能引发过多的系统调用且效率不佳
* 数据紧凑存储，因此可获得更小的索引和更快的全表扫描性能

**3.MEMORY 存储引擎**

提供内存表，也不支持事务和外键。显著提高访问数据的速度，可用于缓存会频繁访问的、可以重构的数据、计算结果、统计值、中间结果。

缺点如下：

* 使用表级锁，虽然内存访问快，但如果频繁的读写，表级锁会成为瓶颈
* 只支持固定大小的行。Varchar类型的字段会存储为固定长度的Char类型，浪费空间
* 不支持TEXT、BLOB字段。当有些查询需要使用到临时表（使用的也是MEMORY存储引擎）时，如果表中有TEXT、BLOB字段，那么会转换为基于磁盘的MyISAM表，严重降低性能
* 由于内存资源成本昂贵，一般不建议设置过大的内存表，如果内存表满了，可通过清除数据或调整内存表参数来避免报错
* 服务器重启后数据会丢失，复制维护时需要小心

### MySQL存储引擎MyISAM与InnoDB如何选择

**1.两种存储引擎的大致区别表现在：**

1）InnoDB支持事务，MyISAM不支持，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而MyISAM就不可以了。

2）MyISAM适合查询以及插入为主的应用，InnoDB适合频繁修改以及涉及到安全性较高的应用

3）InnoDB支持外键，MyISAM不支持

4）从MySQL5.5.5以后，InnoDB是默认引擎

5）InnoDB不支持FULLTEXT类型的索引

6）InnoDB中不保存表的行数，如select count(\*) from table时，InnoDB需要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count(\*)语句包含where条件时MyISAM也需要扫描整个表。

7）对于自增长的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中可以和其他字段一起建立联合索引。

8）清空整个表时，InnoDB是一行一行的删除，效率非常慢。MyISAM则会重建表。

9）InnoDB支持行锁（某些情况下还是锁整表，如 update table set a=1 where user like ‘%lee%’

有人说MYISAM只能用于小型应用，其实这只是一种偏见。

如果数据量比较大，这是需要通过升级架构来解决，比如分表分库，读写分离，而不是单纯地依赖存储引擎。

现在一般都是选用InnoDB了，主要是MyISAM的全表锁，读写串行问题，并发效率锁表，效率低，MyISAM对于读写密集型应用一般是不会去选用的。

**总之：**

1.MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持。

2.MyISAM类型的表强调的是性能，其执行速度比InnoDB类型更快，但是不提供事务支持，而InnoDB提供事务支持已经外部键等高级数据库功能。

如果你还想了解mysql的各种锁：表锁、行锁、乐观锁等，可以点击查看:

[阿里P8架构师谈：MySQL行锁、表锁、悲观锁、乐观锁的特点与应用](http://youzhixueyuan.com/mysql-row-locks-table-locks-pessimistic-locks.html)

## [分布式锁的3种实现（数据库、缓存、Zookeeper)](http://youzhixueyuan.com/3-implementations-of-distributed-locks.html)

### 分布式锁的几种实现方式

目前几乎很多大型网站及应用都是分布式部署的，分布式场景中的数据一致性问题一直是一个比较重要的话题。

分布式的CAP理论告诉我们，任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。

所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

在很多场景中，我们为了保证数据的最终一致性，需要很多的技术方案来支持，比如分布式事务、分布式锁等。有的时候，我们需要保证一个方法在同一时间内只能被同一个线程执行。在单机环境中，Java中其实提供了很多并发处理相关的API，但是这些API在分布式场景中就无能为力了。

也就是说单纯的Java Api并不能提供分布式锁的能力。所以针对分布式锁的实现目前有多种方案。

**针对分布式锁的实现，目前比较常用的有以下几种方案：**

**1.数据库实现**

**2.基于缓存（redis，memcached等）实现**

**3.Zookeeper实现分布式锁**

在分析这几种实现方案之前我们先来想一下，我们需要的分布式锁应该是怎么样的？（这里以方法锁为例，资源锁同理）

1）可以保证在分布式部署的应用集群中，同一个方法在同一时间只能被一台机器上的一个线程执行。

2）这把锁要是一把可重入锁（避免死锁）

3）这把锁最好是一把阻塞锁（根据业务需求考虑要不要这条）

4）有高可用的获取锁和释放锁功能

5）获取锁和释放锁的性能要好

### 基于数据库实现分布式锁

#### 1.基于数据库表

要实现分布式锁，最简单的方式可能就是直接创建一张锁表，然后通过操作该表中的数据来实现了。

当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候就删除这条记录。

创建这样一张数据库表：



当我们想要锁住某个方法时，执行以下SQL：

阿里P8架构师谈：分布式锁的3种实现（数据库、缓存、Zookeeper)

因为我们对method\_name做了唯一性约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，可以执行方法体内容。

当方法执行完毕之后，想要释放锁的话，需要执行以下Sql:

阿里P8架构师谈：分布式锁的3种实现（数据库、缓存、Zookeeper)

**上面这种简单的实现有以下几个问题：**

1、这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。

2、这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。

3、这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。

4、这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

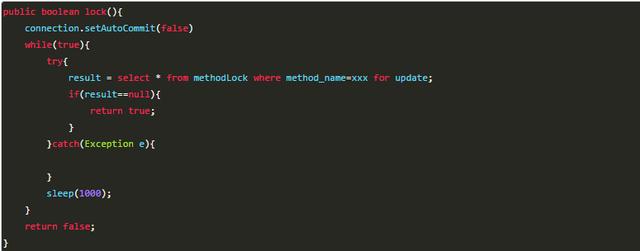
当然，我们也可以有其他方式解决上面的问题。

* 数据库是单点？搞两个数据库，数据之前双向同步。一旦挂掉快速切换到备库上。
* 没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。
* 非阻塞的？搞一个while循环，直到insert成功再返回成功。
* 非重入的？在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。

#### 2.基于数据库排他锁

除了可以通过增删操作数据表中的记录以外，其实还可以借助数据中自带的锁来实现分布式的锁。

我们还用刚刚创建的那张数据库表。可以通过数据库的排他锁来实现分布式锁。 基于MySql的InnoDB引擎，可以使用以下方法来实现加锁操作：



在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁（这里再多提一句，InnoDB引擎在加锁的时候，只有通过索引进行检索的时候才会使用行级锁，否则会使用表级锁。这里我们希望使用行级锁，就要给method\_name添加索引，值得注意的是，这个索引一定要创建成唯一索引，否则会出现多个重载方法之间无法同时被访问的问题。重载方法的话建议把参数类型也加上。）。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。

我们可以认为获得排它锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，再通过以下方法解锁：

阿里P8架构师谈：分布式锁的3种实现（数据库、缓存、Zookeeper)

通过connection.commit()操作来释放锁。

这种方法可以有效的解决上面提到的无法释放锁和阻塞锁的问题。

* 阻塞锁？ for update语句会在执行成功后立即返回，在执行失败时一直处于阻塞状态，直到成功。
* 锁定之后服务宕机，无法释放？使用这种方式，服务宕机之后数据库会自己把锁释放掉。

但是还是无法直接解决数据库单点和可重入问题。

这里还可能存在另外一个问题，虽然我们对method\_name 使用了唯一索引，并且显示使用for update来使用行级锁。但是，MySql会对查询进行优化，即便在条件中使用了索引字段，但是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，如果 MySQL 认为全表扫效率更高，比如对一些很小的表，它就不会使用索引，这种情况下 InnoDB 将使用表锁，而不是行锁。如果发生这种情况就悲剧了。。。

还有一个问题，就是我们要使用排他锁来进行分布式锁的lock，那么一个排他锁长时间不提交，就会占用数据库连接。一旦类似的连接变得多了，就可能把数据库连接池撑爆

#### 数据库实现分布式锁总结

总结一下使用数据库来实现分布式锁的方式，这两种方式都是依赖数据库的一张表，一种是通过表中的记录的存在情况确定当前是否有锁存在，另外一种是通过数据库的排他锁来实现分布式锁。

**数据库实现分布式锁的优点**

* 直接借助数据库，容易理解。

**数据库实现分布式锁的缺点**

* 会有各种各样的问题，在解决问题的过程中会使整个方案变得越来越复杂。
* 操作数据库需要一定的开销，性能问题需要考虑。
* 使用数据库的行级锁并不一定靠谱，尤其是当我们的锁表并不大的时候。

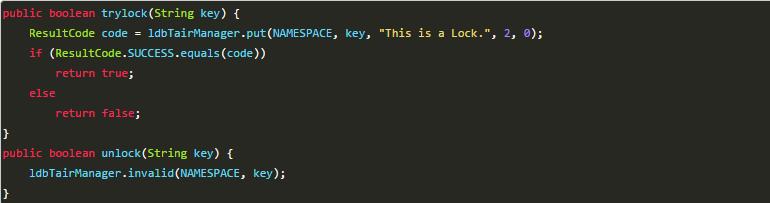
### 基于缓存实现分布式锁

相比较于基于数据库实现分布式锁的方案来说，基于缓存来实现在性能方面会表现的更好一点。而且很多缓存是可以集群部署的，可以解决单点问题。

目前有很多成熟的缓存产品，包括Redis，memcached以及我们公司内部的Tair。

这里以Tair为例来分析下使用缓存实现分布式锁的方案。关于Redis和memcached在网络上有很多相关的文章，并且也有一些成熟的框架及算法可以直接使用。

基于Tair的实现分布式锁其实和Redis类似，其中主要的实现方式是使用TairManager.put方法来实现。



**以上实现方式同样存在几个问题：**

1、这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在tair中，其他线程无法再获得到锁。

2、这把锁只能是非阻塞的，无论成功还是失败都直接返回。

3、这把锁是非重入的，一个线程获得锁之后，在释放锁之前，无法再次获得该锁，因为使用到的key在tair中已经存在。无法再执行put操作。

当然，同样有方式可以解决。

* 没有失效时间？tair的put方法支持传入失效时间，到达时间之后数据会自动删除。
* 非阻塞？while重复执行。
* 非可重入？在一个线程获取到锁之后，把当前主机信息和线程信息保存起来，下次再获取之前先检查自己是不是当前锁的拥有者。

但是，失效时间我设置多长时间为好？如何设置的失效时间太短，方法没等执行完，锁就自动释放了，那么就会产生并发问题。如果设置的时间太长，其他获取锁的线程就可能要平白的多等一段时间。这个问题使用数据库实现分布式锁同样存在

**缓存实现分布式锁总结**

可以使用缓存来代替数据库来实现分布式锁，这个可以提供更好的性能，同时，很多缓存服务都是集群部署的，可以避免单点问题。并且很多缓存服务都提供了可以用来实现分布式锁的方法，比如Tair的put方法，redis的setnx方法等。并且，这些缓存服务也都提供了对数据的过期自动删除的支持，可以直接设置超时时间来控制锁的释放。

**缓存实现分布式锁的优点**

* 性能好，实现起来较为方便。

**缓存实现分布式锁的缺点**

* 通过超时时间来控制锁的失效时间并不是十分的靠谱。

### 基于Zookeeper实现分布式锁

基于zookeeper临时有序节点可以实现的分布式锁。

大致思想即为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

**来看下Zookeeper能不能解决前面提到的问题。**

* 锁无法释放？使用Zookeeper可以有效的解决锁无法释放的问题，因为在创建锁的时候，客户端会在ZK中创建一个临时节点，一旦客户端获取到锁之后突然挂掉（Session连接断开），那么这个临时节点就会自动删除掉。其他客户端就可以再次获得锁。
* 非阻塞锁？使用Zookeeper可以实现阻塞的锁，客户端可以通过在ZK中创建顺序节点，并且在节点上绑定监听器，一旦节点有变化，Zookeeper会通知客户端，客户端可以检查自己创建的节点是不是当前所有节点中序号最小的，如果是，那么自己就获取到锁，便可以执行业务逻辑了。
* 不可重入？使用Zookeeper也可以有效的解决不可重入的问题，客户端在创建节点的时候，把当前客户端的主机信息和线程信息直接写入到节点中，下次想要获取锁的时候和当前最小的节点中的数据比对一下就可以了。如果和自己的信息一样，那么自己直接获取到锁，如果不一样就再创建一个临时的顺序节点，参与排队。
* 单点问题？使用Zookeeper可以有效的解决单点问题，ZK是集群部署的，只要集群中有半数以上的机器存活，就可以对外提供服务。

可以直接使用zookeeper第三方库Curator客户端，这个客户端中封装了一个可重入的锁服务。



Curator提供的InterProcessMutex是分布式锁的实现。acquire方法用户获取锁，release方法用于释放锁。

使用ZK实现的分布式锁好像完全符合了本文开头我们对一个分布式锁的所有期望。但是，其实并不是，Zookeeper实现的分布式锁其实存在一个缺点，那就是性能上可能并没有缓存服务那么高。因为每次在创建锁和释放锁的过程中，都要动态创建、销毁瞬时节点来实现锁功能。ZK中创建和删除节点只能通过Leader服务器来执行，然后将数据同不到所有的Follower机器上。

其实，使用Zookeeper也有可能带来并发问题，只是并不常见而已。考虑这样的情况，由于网络抖动，客户端可ZK集群的session连接断了，那么zk以为客户端挂了，就会删除临时节点，这时候其他客户端就可以获取到分布式锁了。就可能产生并发问题。这个问题不常见是因为zk有重试机制，一旦zk集群检测不到客户端的心跳，就会重试，Curator客户端支持多种重试策略。多次重试之后还不行的话才会删除临时节点。（所以，选择一个合适的重试策略也比较重要，要在锁的粒度和并发之间找一个平衡。）

**Zookeeper实现分布式锁总结**

Zookeeper实现分布式锁的优点

* 有效的解决单点问题，不可重入问题，非阻塞问题以及锁无法释放的问题。实现起来较为简单。

Zookeeper实现分布式锁的缺点

* 性能上不如使用缓存实现分布式锁。 需要对ZK的原理有所了解。

### 三种方案的比较

上面几种方式，哪种方式都无法做到完美。就像CAP一样，在复杂性、可靠性、性能等方面无法同时满足，所以，根据不同的应用场景选择最适合自己的才是王道。

**1.从理解的难易程度角度（从低到高）**

数据库 > 缓存 > Zookeeper

**2.从实现的复杂性角度（从低到高）**

Zookeeper >= 缓存 > 数据库

**3.从性能角度（从高到低）**

缓存 > Zookeeper >= 数据库

**4.从可靠性角度（从高到低）**

Zookeeper > 缓存 > 数据库

## MySQL数据库的索引原理、与慢SQL优化的5大原则

MySQL凭借着出色的性能、低廉的成本、丰富的资源，已经成为绝大多数互联网公司的首选关系型数据库。虽然性能出色，但所谓“好马配好鞍”，如何能够更好的使用它，已经成为开发工程师的必修课，我们经常会从职位描述上看到诸如“精通MySQL”、“SQL语句优化”、“了解数据库原理”等要求。

我们知道一般的应用系统，读写比例在10:1左右，而且插入操作和一般的更新操作很少出现性能问题，遇到最多的，也是最容易出问题的，还是一些复杂的查询操作，所以查询语句的优化显然是重中之重。

本文旨在以开发工程师的角度来解释数据库索引的原理和如何优化慢查询。

### MySQL索引原理

**1.索引目的**

索引的目的在于提高查询效率，可以类比字典，如果要查“mysql”这个单词，我们肯定需要定位到m字母，然后从下往下找到y字母，再找到剩下的sql。如果没有索引，那么你可能需要把所有单词看一遍才能找到你想要的，如果我想找到m开头的单词呢？或者ze开头的单词呢？是不是觉得如果没有索引，这个事情根本无法完成？

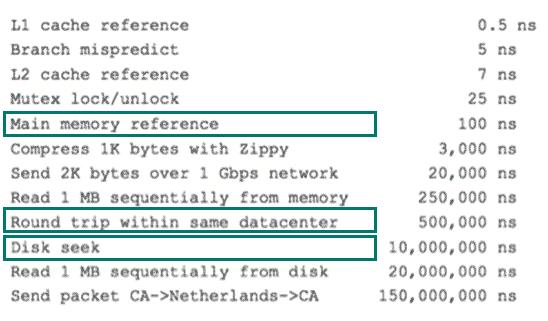
### 2.索引原理

除了词典，生活中随处可见索引的例子，如火车站的车次表、图书的目录等。它们的原理都是一样的，通过不断的缩小想要获得数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件，也就是我们总是通过同一种查找方式来锁定数据。

数据库也是一样，但显然要复杂许多，因为不仅面临着等值查询，还有范围查询(>、<、between、in)、模糊查询(like)、并集查询(or)等等。数据库应该选择怎么样的方式来应对所有的问题呢？我们回想字典的例子，能不能把数据分成段，然后分段查询呢？最简单的如果1000条数据，1到100分成第一段，101到200分成第二段，201到300分成第三段……这样查第250条数据，只要找第三段就可以了，一下子去除了90%的无效数据。但如果是1千万的记录呢，分成几段比较好？稍有算法基础的同学会想到搜索树，其平均复杂度是lgN，具有不错的查询性能。但这里我们忽略了一个关键的问题，复杂度模型是基于每次相同的操作成本来考虑的，数据库实现比较复杂，数据保存在磁盘上，而为了提高性能，每次又可以把部分数据读入内存来计算，因为我们知道访问磁盘的成本大概是访问内存的十万倍左右，所以简单的搜索树难以满足复杂的应用场景。

**3.磁盘IO与预读**

前面提到了访问磁盘，那么这里先简单介绍一下磁盘IO和预读，磁盘读取数据靠的是机械运动，每次读取数据花费的时间可以分为寻道时间、旋转延迟、传输时间三个部分，寻道时间指的是磁臂移动到指定磁道所需要的时间，主流磁盘一般在5ms以下；旋转延迟就是我们经常听说的磁盘转速，比如一个磁盘7200转，表示每分钟能转7200次，也就是说1秒钟能转120次，旋转延迟就是1/120/2 = 4.17ms；传输时间指的是从磁盘读出或将数据写入磁盘的时间，一般在零点几毫秒，相对于前两个时间可以忽略不计。那么访问一次磁盘的时间，即一次磁盘IO的时间约等于5+4.17 = 9ms左右，听起来还挺不错的，但要知道一台500 -MIPS的机器每秒可以执行5亿条指令，因为指令依靠的是电的性质，换句话说执行一次IO的时间可以执行40万条指令，数据库动辄十万百万乃至千万级数据，每次9毫秒的时间，显然是个灾难。下图是计算机硬件延迟的对比图，供大家参考：

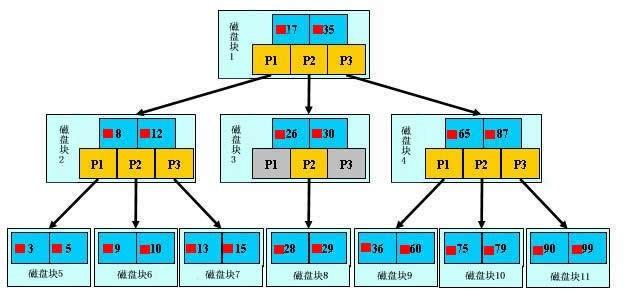


考虑到磁盘IO是非常高昂的操作，计算机操作系统做了一些优化，当一次IO时，不光把当前磁盘地址的数据，而是把相邻的数据也都读取到内存缓冲区内，因为局部预读性原理告诉我们，当计算机访问一个地址的数据的时候，与其相邻的数据也会很快被访问到。每一次IO读取的数据我们称之为一页(page)。具体一页有多大数据跟操作系统有关，一般为4k或8k，也就是我们读取一页内的数据时候，实际上才发生了一次IO，这个理论对于索引的数据结构设计非常有帮助。

**4.索引的数据结构**

前面讲了生活中索引的例子，索引的基本原理，数据库的复杂性，又讲了操作系统的相关知识，目的就是让大家了解，任何一种数据结构都不是凭空产生的，一定会有它的背景和使用场景，我们现在总结一下，我们需要这种数据结构能够做些什么，其实很简单，那就是：每次查找数据时把磁盘IO次数控制在一个很小的数量级，最好是常数数量级。那么我们就想到如果一个高度可控的多路搜索树是否能满足需求呢？就这样，b+树应运而生。

**5.详解b+树**



如上图，是一颗b+树，关于b+树的定义可以参见B+树，这里只说一些重点，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。

**6.b+树的查找过程**

如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

**7.b+树性质**

1.通过上面的分析，我们知道IO次数取决于b+数的高度h，假设当前数据表的数据为N，每个磁盘块的数据项的数量是m，则有h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。这也是为什么b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于1时将会退化成线性表。

2.当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

#### 慢查询优化

关于MySQL索引原理是比较枯燥的东西，大家只需要有一个感性的认识，并不需要理解得非常透彻和深入。我们回头来看看一开始我们说的慢查询，了解完索引原理之后，大家是不是有什么想法呢？先总结一下索引的几大基本原则

**建索引的几大原则**

**1.最左前缀匹配原则**

非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

**2.=和in可以乱序**

比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

**3.尽量选择区分度高的列作为索引**

区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录

**4.索引列不能参与计算，保持列“干净”**

比如from\_unixtime(create\_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成create\_time = unix\_timestamp(’2014-05-29’);

**5.尽量的扩展索引，不要新建索引。**

比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可

**查询优化神器 – explain命令**

关于explain命令相信大家并不陌生，具体用法和字段含义可以参考官网explain-output，这里需要强调rows是核心指标，绝大部分rows小的语句执行一定很快（有例外，下面会讲到）。所以优化语句基本上都是在优化rows。

### 慢查询优化基本步骤

0.先运行看看是否真的很慢，注意设置SQL\_NO\_CACHE

1.where条件单表查，锁定最小返回记录表。这句话的意思是把查询语句的where都应用到表中返回的记录数最小的表开始查起，单表每个字段分别查询，看哪个字段的区分度最高

2.explain查看执行计划，是否与1预期一致（从锁定记录较少的表开始查询）

3.order by limit 形式的sql语句让排序的表优先查

4.了解业务方使用场景

5.加索引时参照建索引的几大原则

6.观察结果，不符合预期继续从0分析

#### 慢查询案例

下面几个例子详细解释了如何分析和优化慢查询

**复杂语句写法**

很多情况下，我们写SQL只是为了实现功能，这只是第一步，不同的语句书写方式对于效率往往有本质的差别，这要求我们对mysql的执行计划和索引原则有非常清楚的认识，请看下面的语句

select

distinct cert.emp\_id

from

cm\_log cl

inner join

(

select

emp.id as emp\_id,

emp\_cert.id as cert\_id

from

employee emp

left join

emp\_certificate emp\_cert

on emp.id = emp\_cert.emp\_id

where

emp.is\_deleted=0

) cert

on (

cl.ref\_table='Employee'

and cl.ref\_oid= cert.emp\_id

)

or (

cl.ref\_table='EmpCertificate'

and cl.ref\_oid= cert.cert\_id

)

where

cl.last\_upd\_date >='2013-11-07 15:03:00'

and cl.last\_upd\_date<='2013-11-08 16:00:00';

0.先运行一下，53条记录 1.87秒，又没有用聚合语句，比较慢

53 rows in set (1.87 sec)

1.explain

+----+-------------+------------+-------+---------------------------------+-----------------------+---------+-------------------+-------+--------------------------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+------------+-------+---------------------------------+-----------------------+---------+-------------------+-------+--------------------------------+

| 1 | PRIMARY | cl | range | cm\_log\_cls\_id,idx\_last\_upd\_date | idx\_last\_upd\_date | 8 | NULL | 379 | Using where; Using temporary |

| 1 | PRIMARY | <derived2> | ALL | NULL | NULL | NULL | NULL | 63727 | Using where; Using join buffer |

| 2 | DERIVED | emp | ALL | NULL | NULL | NULL | NULL | 13317 | Using where |

| 2 | DERIVED | emp\_cert | ref | emp\_certificate\_empid | emp\_certificate\_empid | 4 | meituanorg.emp.id | 1 | Using index |

+----+-------------+------------+-------+---------------------------------+-----------------------+---------+-------------------+-------+--------------------------------+

简述一下执行计划，首先mysql根据idx\_last\_upd\_date索引扫描cm\_log表获得379条记录；然后查表扫描了63727条记录，分为两部分，derived表示构造表，也就是不存在的表，可以简单理解成是一个语句形成的结果集，后面的数字表示语句的ID。derived2表示的是ID = 2的查询构造了虚拟表，并且返回了63727条记录。我们再来看看ID = 2的语句究竟做了写什么返回了这么大量的数据，首先全表扫描employee表13317条记录，然后根据索引emp\_certificate\_empid关联emp\_certificate表，rows = 1表示，每个关联都只锁定了一条记录，效率比较高。获得后，再和cm\_log的379条记录根据规则关联。从执行过程上可以看出返回了太多的数据，返回的数据绝大部分cm\_log都用不到，因为cm\_log只锁定了379条记录。

如何优化呢？可以看到我们在运行完后还是要和cm\_log做join,那么我们能不能之前和cm\_log做join呢？仔细分析语句不难发现，其基本思想是如果cm\_log的ref\_table是EmpCertificate就关联emp\_certificate表，如果ref\_table是Employee就关联employee表，我们完全可以拆成两部分，并用union连接起来，注意这里用union，而不用union all是因为原语句有“distinct”来得到唯一的记录，而union恰好具备了这种功能。如果原语句中没有distinct不需要去重，我们就可以直接使用union all了，因为使用union需要去重的动作，会影响SQL性能。

优化过的语句如下

select

emp.id

from

cm\_log cl

inner join

employee emp

on cl.ref\_table = 'Employee'

and cl.ref\_oid = emp.id

where

cl.last\_upd\_date >='2013-11-07 15:03:00'

and cl.last\_upd\_date<='2013-11-08 16:00:00'

and emp.is\_deleted = 0

union

select

emp.id

from

cm\_log cl

inner join

emp\_certificate ec

on cl.ref\_table = 'EmpCertificate'

and cl.ref\_oid = ec.id

inner join

employee emp

on emp.id = ec.emp\_id

where

cl.last\_upd\_date >='2013-11-07 15:03:00'

and cl.last\_upd\_date<='2013-11-08 16:00:00'

and emp.is\_deleted = 0

4.不需要了解业务场景，只需要改造的语句和改造之前的语句保持结果一致

5.现有索引可以满足，不需要建索引

6.用改造后的语句实验一下，只需要10ms 降低了近200倍！

+----+--------------+------------+--------+---------------------------------+-------------------+---------+-----------------------+------+-------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+--------------+------------+--------+---------------------------------+-------------------+---------+-----------------------+------+-------------+

| 1 | PRIMARY | cl | range | cm\_log\_cls\_id,idx\_last\_upd\_date | idx\_last\_upd\_date | 8 | NULL | 379 | Using where |

| 1 | PRIMARY | emp | eq\_ref | PRIMARY | PRIMARY | 4 | meituanorg.cl.ref\_oid | 1 | Using where |

| 2 | UNION | cl | range | cm\_log\_cls\_id,idx\_last\_upd\_date | idx\_last\_upd\_date | 8 | NULL | 379 | Using where |

| 2 | UNION | ec | eq\_ref | PRIMARY,emp\_certificate\_empid | PRIMARY | 4 | meituanorg.cl.ref\_oid | 1 | |

| 2 | UNION | emp | eq\_ref | PRIMARY | PRIMARY | 4 | meituanorg.ec.emp\_id | 1 | Using where |

| NULL | UNION RESULT | <union1,2> | ALL | NULL | NULL | NULL | NULL | NULL | |

+----+--------------+------------+--------+---------------------------------+-------------------+---------+-----------------------+------+-------------+

53 rows in set (0.01 sec)

**明确应用场景**

举这个例子的目的在于颠覆我们对列的区分度的认知，一般上我们认为区分度越高的列，越容易锁定更少的记录，但在一些特殊的情况下，这种理论是有局限性的

select

\*

from

stage\_poi sp

where

sp.accurate\_result=1

and (

sp.sync\_status=0

or sp.sync\_status=2

or sp.sync\_status=4

);

0.先看看运行多长时间,951条数据6.22秒，真的很慢

951 rows in set (6.22 sec)

1.先explain，rows达到了361万，type = ALL表明是全表扫描

+----+-------------+-------+------+---------------+------+---------+------+---------+-------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+------+---------------+------+---------+------+---------+-------------+

| 1 | SIMPLE | sp | ALL | NULL | NULL | NULL | NULL | 3613155 | Using where |

+----+-------------+-------+------+---------------+------+---------+------+---------+-------------+

2.所有字段都应用查询返回记录数，因为是单表查询 0已经做过了951条

3.让explain的rows 尽量逼近951

看一下accurate\_result = 1的记录数

select count(\*),accurate\_result from stage\_poi group by accurate\_result;

+----------+-----------------+

| count(\*) | accurate\_result |

+----------+-----------------+

| 1023 | -1 |

| 2114655 | 0 |

| 972815 | 1 |

+----------+-----------------+

我们看到accurate\_result这个字段的区分度非常低，整个表只有-1,0,1三个值，加上索引也无法锁定特别少量的数据

再看一下sync\_status字段的情况

select count(\*),sync\_status from stage\_poi group by sync\_status;

+----------+-------------+

| count(\*) | sync\_status |

+----------+-------------+

| 3080 | 0 |

| 3085413 | 3 |

+----------+-------------+

同样的区分度也很低，根据理论，也不适合建立索引

问题分析到这，好像得出了这个表无法优化的结论，两个列的区分度都很低，即便加上索引也只能适应这种情况，很难做普遍性的优化，比如当sync\_status 0、3分布的很平均，那么锁定记录也是百万级别的

4.找业务方去沟通，看看使用场景。业务方是这么来使用这个SQL语句的，每隔五分钟会扫描符合条件的数据，处理完成后把sync\_status这个字段变成1,五分钟符合条件的记录数并不会太多，1000个左右。了解了业务方的使用场景后，优化这个SQL就变得简单了，因为业务方保证了数据的不平衡，如果加上索引可以过滤掉绝大部分不需要的数据

5.根据建立索引规则，使用如下语句建立索引

alter table stage\_poi add index idx\_acc\_status(accurate\_result,sync\_status);

6.观察预期结果,发现只需要200ms，快了30多倍。

952 rows in set (0.20 sec)

我们再来回顾一下分析问题的过程，单表查询相对来说比较好优化，大部分时候只需要把where条件里面的字段依照规则加上索引就好，如果只是这种“无脑”优化的话，显然一些区分度非常低的列，不应该加索引的列也会被加上索引，这样会对插入、更新性能造成严重的影响，同时也有可能影响其它的查询语句。

所以我们第4步调差SQL的使用场景非常关键，我们只有知道这个业务场景，才能更好地辅助我们更好的分析和优化查询语句。

慢查询的案例就分析到这儿，以上只是一些比较典型的案例。

我们在优化过程中遇到过超过1000行，涉及到16个表join的“垃圾SQL”，也遇到过线上线下数据库差异导致应用直接被慢查询拖死，也遇到过varchar等值比较没有写单引号，还遇到过笛卡尔积查询直接把从库搞死。再多的案例其实也只是一些经验的积累，如果我们熟悉查询优化器、索引的内部原理，那么分析这些案例就变得特别简单了。

## [分布式事务的解决方案，以及原理、总结](http://youzhixueyuan.com/solution-and-summary-of-distributed-transaction.html)

分布式事务是企业集成中的一个技术难点，也是每一个分布式系统架构中都会涉及到的一个东西，特别是在这几年越来越火的微服务架构中，几乎可以说是无法避免，本文就围绕分布式事务各方面与大家进行介绍。

### 事务

**1.1 什么是事务**

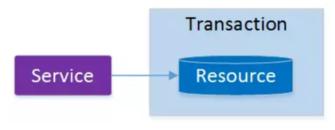
数据库事务(简称：事务，Transaction)是指数据库执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。

**事务拥有以下四个特性，习惯上被称为ACID特性：**

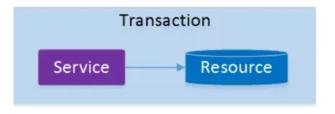
* **原子性(Atomicity)**：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。
* **一致性(Consistency)**：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态是指数据库中的数据应满足完整性约束。除此之外，一致性还有另外一层语义，就是事务的中间状态不能被观察到(这层语义也有说应该属于原子性)。
* **隔离性(Isolation)**：多个事务并发执行时，一个事务的执行不应影响其他事务的执行，如同只有这一个操作在被数据库所执行一样。
* **持久性(Durability)**：已被提交的事务对数据库的修改应该永久保存在数据库中。在事务结束时，此操作将不可逆转。

**1.2 本地事务**

起初，事务仅限于对单一数据库资源的访问控制：



架构服务化以后，事务的概念延伸到了服务中。倘若将一个单一的服务操作作为一个事务，那么整个服务操作只能涉及一个单一的数据库资源：

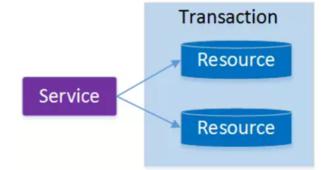


**这类基于单个服务单一数据库资源访问的事务，被称为本地事务(Local Transaction)。**

### 分布式事务

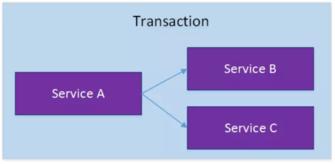
本地事务主要限制在单个会话内，不涉及多个数据库资源。但是在基于SOA(Service-Oriented Architecture，面向服务架构)的分布式应用环境下，越来越多的应用要求对**多个数据库资源，多个服务的访问都能纳入到同一个事务当中，分布式事务应运而生**。

最早的分布式事务应用架构很简单，不涉及服务间的访问调用，仅仅是服务内操作涉及到对多个数据库资源的访问。

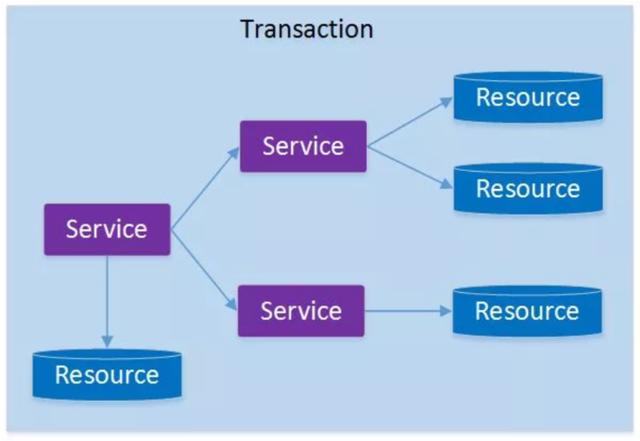


当一个服务操作访问不同的数据库资源，又希望对它们的访问具有事务特性时，就需要采用分布式事务来协调所有的事务参与者。

对于上面介绍的分布式事务应用架构，尽管一个服务操作会访问多个数据库资源，但是毕竟整个事务还是控制在单一服务的内部。如果一个服务操作需要调用另外一个服务，这时的事务就需要跨越多个服务了。在这种情况下，起始于某个服务的事务在调用另外一个服务的时候，需要以某种机制流转到另外一个服务，从而使被调用的服务访问的资源也自动加入到该事务当中来。下图反映了这样一个跨越多个服务的分布式事务：



如果将上面这两种场景(一个服务可以调用多个数据库资源，也可以调用其他服务)结合在一起，对此进行延伸，整个分布式事务的参与者将会组成如下图所示的树形拓扑结构。在一个跨服务的分布式事务中，事务的发起者和提交均系同一个，它可以是整个调用的客户端，也可以是客户端最先调用的那个服务。



较之基于单一数据库资源访问的本地事务，分布式事务的应用架构更为复杂。

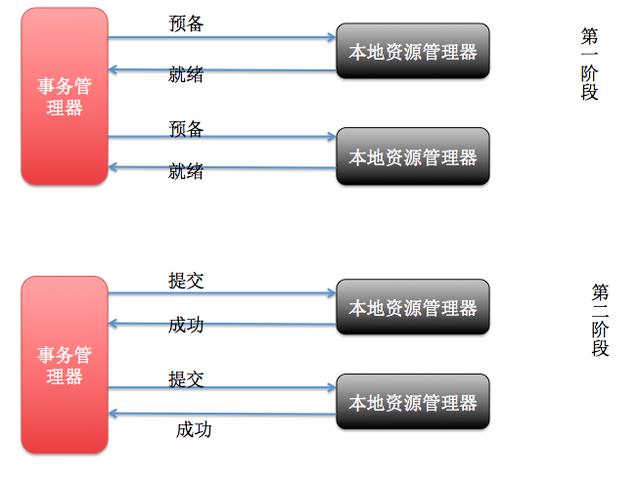
在不同的分布式应用架构下，实现一个分布式事务要考虑的问题并不完全一样，比如对多资源的协调、事务的跨服务传播等，实现机制也是复杂多变。尽管有这么多工程细节需要考虑，但**分布式事务最核心的还是其 ACID 特性**。因此，想要了解一个分布式事务，就先从了解它是怎么实现事务 ACID 特性开始。

下文将从两个最常见的分布式事务模型入手，着重分析分布式事务的基础共通点，即如何保证分布式事务的 ACID 特性。

### 常见的分布式事务解决方案

**1.基于XA协议的两阶段提交**

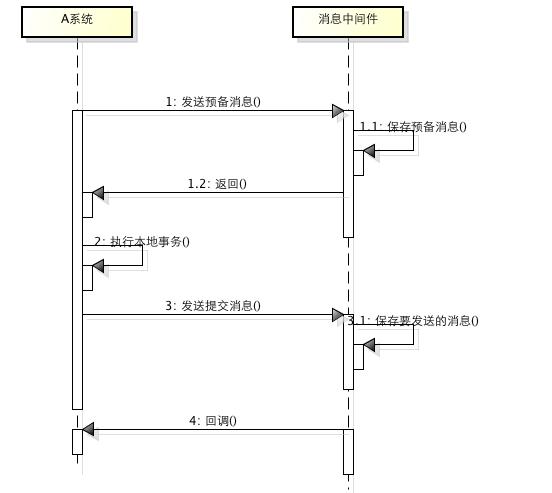
XA是一个分布式事务协议，由Tuxedo提出。XA中大致分为两部分：事务管理器和本地资源管理器。其中本地资源管理器往往由数据库实现，比如Oracle、DB2这些商业数据库都实现了XA接口，而事务管理器作为全局的调度者，负责各个本地资源的提交和回滚。XA实现分布式事务的原理如下：



总的来说，XA协议比较简单，而且一旦商业数据库实现了XA协议，使用分布式事务的成本也比较低。但是，XA也有致命的缺点，那就是性能不理想，特别是在交易下单链路，往往并发量很高，XA无法满足高并发场景。XA目前在商业数据库支持的比较理想，在mysql数据库中支持的不太理想，mysql的XA实现，没有记录prepare阶段日志，主备切换回导致主库与备库数据不一致。许多nosql也没有支持XA，这让XA的应用场景变得非常狭隘。

**2.消息事务+最终一致性**

所谓的消息事务就是基于消息中间件的两阶段提交，本质上是对消息中间件的一种特殊利用，它是将本地事务和发消息放在了一个分布式事务里，保证要么本地操作成功成功并且对外发消息成功，要么两者都失败，开源的RocketMQ就支持这一特性，具体原理如下：



1、A系统向消息中间件发送一条预备消息

2、消息中间件保存预备消息并返回成功

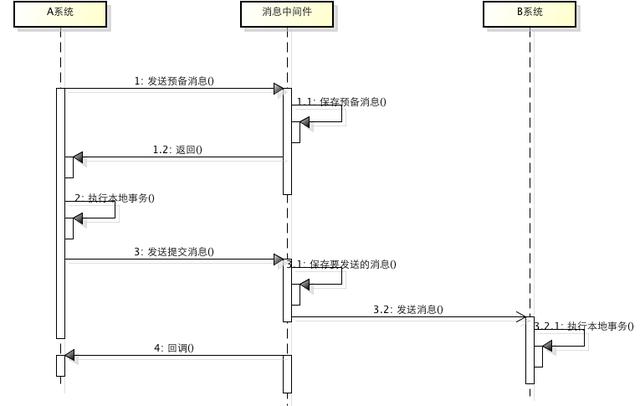
3、A执行本地事务

4、A发送提交消息给消息中间件

通过以上4步完成了一个消息事务。对于以上的4个步骤，每个步骤都可能产生错误，下面一一分析：

* 步骤一出错，则整个事务失败，不会执行A的本地操作
* 步骤二出错，则整个事务失败，不会执行A的本地操作
* 步骤三出错，这时候需要回滚预备消息，怎么回滚？答案是A系统实现一个消息中间件的回调接口，消息中间件会去不断执行回调接口，检查A事务执行是否执行成功，如果失败则回滚预备消息
* 步骤四出错，这时候A的本地事务是成功的，那么消息中间件要回滚A吗？答案是不需要，其实通过回调接口，消息中间件能够检查到A执行成功了，这时候其实不需要A发提交消息了，消息中间件可以自己对消息进行提交，从而完成整个消息事务

基于消息中间件的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A系统的本地操作+发消息）+B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发出来了，这时候B会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到B操作成功，这样就变相地实现了A与B的分布式事务。原理如下：



虽然上面的方案能够完成A和B的操作，但是A和B并不是严格一致的，而是最终一致的，我们在这里牺牲了一致性，换来了性能的大幅度提升。当然，这种玩法也是有风险的，如果B一直执行不成功，那么一致性会被破坏，具体要不要玩，还是得看业务能够承担多少风险。

**3.TCC 模型**

TCC(Try-Confirm-Cancel)分布式事务模型相对于 XA 等传统模型，其特征在于它不依赖资源管理器(RM)对分布式事务的支持，而是通过对业务逻辑的分解来实现分布式事务。

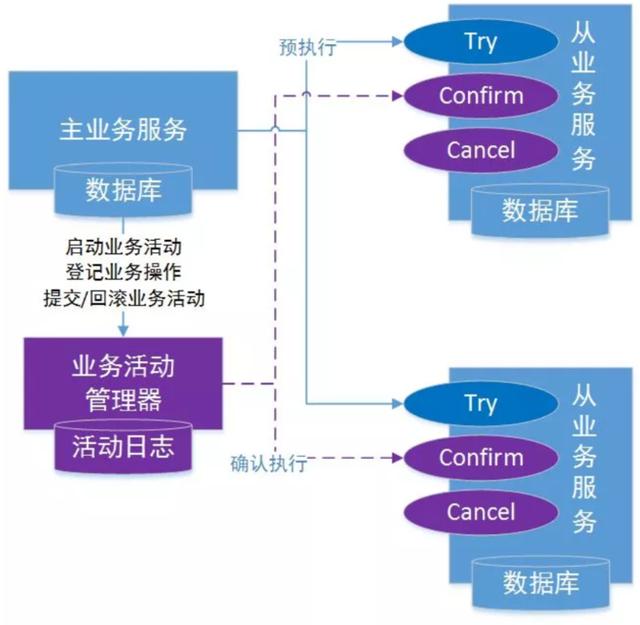
TCC 模型认为对于业务系统中一个特定的业务逻辑，其对外提供服务时，必须接受一些不确定性，即对业务逻辑初步操作的调用仅是一个临时性操作，调用它的主业务服务保留了后续的取消权。如果主业务服务认为全局事务应该回滚，它会要求取消之前的临时性操作，这就对应从业务服务的取消操作。而当主业务服务认为全局事务应该提交时，它会放弃之前临时性操作的取消权，这对应从业务服务的确认操作。每一个初步操作，最终都会被确认或取消。

因此，针对一个具体的业务服务，TCC 分布式事务模型需要业务系统提供三段业务逻辑：

初步操作 Try：完成所有业务检查，预留必须的业务资源。

确认操作 Confirm：真正执行的业务逻辑，不作任何业务检查，只使用 Try 阶段预留的业务资源。因此，只要 Try 操作成功，Confirm 必须能成功。另外，Confirm 操作需满足幂等性，保证一笔分布式事务有且只能成功一次。

取消操作 Cancel：释放 Try 阶段预留的业务资源。同样的，Cancel 操作也需要满足幂等性。



**TCC 分布式事务模型包括三部分**：

**1.主业务服务**：主业务服务为整个业务活动的发起方，服务的编排者，负责发起并完成整个业务活动。

**2.从业务服务**：从业务服务是整个业务活动的参与方，负责提供 TCC 业务操作，实现初步操作(Try)、确认操作(Confirm)、取消操作(Cancel)三个接口，供主业务服务调用。

**3.业务活动管理器**：业务活动管理器管理控制整个业务活动，包括记录维护 TCC 全局事务的事务状态和每个从业务服务的子事务状态，并在业务活动提交时调用所有从业务服务的 Confirm 操作，在业务活动取消时调用所有从业务服务的 Cancel 操作。

**一个完整的 TCC 分布式事务流程如下：**

1. 主业务服务首先开启本地事务;
2. 主业务服务向业务活动管理器申请启动分布式事务主业务活动;
3. 然后针对要调用的从业务服务，主业务活动先向业务活动管理器注册从业务活动，然后调用从业务服务的 Try 接口;
4. 当所有从业务服务的 Try 接口调用成功，主业务服务提交本地事务;若调用失败，主业务服务回滚本地事务;
5. 若主业务服务提交本地事务，则 TCC 模型分别调用所有从业务服务的 Confirm 接口;若主业务服务回滚本地事务，则分别调用 Cancel 接口;
6. 所有从业务服务的 Confirm 或 Cancel 操作完成后，全局事务结束。

**TCC模型小结**

所谓的TCC编程模式，也是两阶段提交的一个变种。TCC提供了一个编程框架，将整个业务逻辑分为三块：**Try、Confirm和Cancel**三个操作。以在线下单为例，Try阶段会去扣库存，Confirm阶段则是去更新订单状态，如果更新订单失败，则进入Cancel阶段，会去恢复库存。总之，TCC就是通过代码人为实现了两阶段提交，不同的业务场景所写的代码都不一样，复杂度也不一样，因此，这种模式并不能很好地被复用。

### 分布式事务总结

分布式事务，本质上是对多个数据库的事务进行统一控制，按照控制力度可以分为：不控制、部分控制和完全控制。不控制就是不引入分布式事务，部分控制就是各种变种的两阶段提交，包括上面提到的消息事务+最终一致性、TCC模式，而完全控制就是完全实现两阶段提交。部分控制的好处是并发量和性能很好，缺点是数据一致性减弱了，完全控制则是牺牲了性能，保障了一致性，具体用哪种方式，最终还是取决于业务场景。作为技术人员，一定不能忘了技术是为业务服务的，不要为了技术而技术，针对不同业务进行技术选型也是一种很重要的能力！

## MySQL行锁、表锁、悲观锁、乐观锁的特点与应用

我们在操作数据库的时候，可能会由于并发问题而引起的数据的不一致性（数据冲突）。如何保证数据并发访问的一致性、有效性，是所有数据库必须解决的一个问题，锁的冲突也是影响数据库并发访问性能的一个重要因素，从这一角度来说，锁对于数据库而言就显得尤为重要。

### MySQL锁概述

相对其他数据库而言，MySQL的锁机制比较简单，其最显著的特点是不同的存储引擎支持不同的锁机制。

比如：

1. MyISAM和MEMORY存储引擎采用的是表级锁（table-level locking）；
2. InnoDB存储引擎既支持行级锁（ row-level locking），也支持表级锁，但默认情况下是采用行级锁。

**MySQL主要的两种锁的特性可大致归纳如下:**



* 表级锁： 开销小，加锁快；不会出现死锁(因为MyISAM会一次性获得SQL所需的全部锁)；锁定粒度大，发生锁冲突的概率最高,并发度最低。
* 行级锁： 开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。
* 页锁：开销和加锁速度介于表锁和行锁之间；会出现死锁；锁定粒度介于表锁和行锁之间，并发度一般

### 行锁 和 表锁

**1.**主要是针对锁粒度划分的，一般分为：**行锁、表锁、库锁**

**（1）行锁：访问数据库的时候，锁定整个行数据，防止并发错误。**

**（2）表锁：访问数据库的时候，锁定整个表数据，防止并发错误。**

**2.行锁 和 表锁 的区别：**

* **表锁**： 开销小，加锁快，不会出现死锁；锁定力度大，发生锁冲突概率高，并发度最低
* **行锁**： 开销大，加锁慢，**会出现死锁**；锁定粒度小，发生锁冲突的概率低，并发度高

### 悲观锁 和 乐观锁

**（1）悲观锁：**顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以**每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。**

传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

**（2）乐观锁：** 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在**更新的时候会判断一下**在此期间别人有没有去更新这个数据，可以使用版本号等机制。

**乐观锁适用于多读的应用类型，这样可以提高吞吐量**，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

**（3）悲观锁 和 乐观锁的区别：**

两种锁各有优缺点，不可认为一种好于另一种，像**乐观锁适用于写比较少的情况下**，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但**如果经常产生冲突**，上层应用会不断的进行retry，这样反倒是降低了性能，**所以这种情况下用悲观锁就比较合适**。

### 共享锁

共享锁指的就是对于多个不同的事务，对同一个资源共享同一个锁。相当于对于同一把门，它拥有多个钥匙一样。就像这样，你家有一个大门，大门的钥匙有好几把，你有一把，你女朋友有一把，你们都可能通过这把钥匙进入你们家，这个就是所谓的共享锁。

刚刚说了，对于悲观锁，一般数据库已经实现了，**共享锁也属于悲观锁的一种**，那么共享锁在mysql中是通过什么命令来调用呢。通过查询资料，了解到通过在执行语句后面加上**lock in share mode**就代表对某些资源加上共享锁了。

**什么时候使用表锁**

对于InnoDB表，在绝大部分情况下都应该使用行级锁，因为事务和行锁往往是我们之所以选择InnoDB表的理由。但在个别特殊事务中，也可以考虑使用表级锁。

* 第一种情况是：事务需要更新大部分或全部数据，表又比较大，如果使用默认的行锁，不仅这个事务执行效率低，而且可能造成其他事务长时间锁等待和锁冲突，这种情况下可以考虑使用表锁来提高该事务的执行速度。
* 第二种情况是：事务涉及多个表，比较复杂，很可能引起死锁，造成大量事务回滚。这种情况也可以考虑一次性锁定事务涉及的表，从而避免死锁、减少数据库因事务回滚带来的开销。

当然，应用中这两种事务不能太多，否则，就应该考虑使用MyISAM表了。

### 表锁和行锁应用场景：

* **表级锁使用与并发性不高，以查询为主，少量更新的应用**，比如小型的web应用；
* 而**行级锁适用于高并发环境下，对事务完整性要求较高的系统**，如在线事务处理系统。

## **架构设计之数据库垂直、水平拆分六大原则**

### 数据拆分前其实是要首先做准备工作的，然后才是开始数据拆分，我先讲拆分前需要做的事情：

1. 第一步：采用分布式缓存redis、memcached等降低对数据库的读操作。
2. 第二步：如果缓存使用过后，数据库访问量还是非常大，可以考虑数据库读、写分离原则。
3. 第三步：当我们使用读写分离、缓存后，数据库的压力还是很大的时候，这就需要使用到数据库拆分了。

数据库拆分原则：就是指通过某种特定的条件，按照某个维度，将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）上面以达到分散单库（主机）负载的效果。

### 第一步，首选垂直拆分

一个数据库由很多表的构成，每个表对应着不同的业务，垂直切分是指按照业务将表进行分类，分布到不同的数据库上面，这样也就将数据或者说压力分担到不同的库上面 。

比如淘宝中期开始的数据库端按照业务垂直拆分：按照业务交易数据库、用户数据库、商品数据库、店铺数据库等进行拆分。

采用垂直拆分

优点：

1. 拆分后业务清晰，拆分规则明确。

2. 系统之间整合或扩展容易。

3. 数据维护简单。

缺点：

1. 部分业务表无法join，只能通过接口方式解决，提高了系统复杂度。

2. 受每种业务不同的限制存在单库性能瓶颈，不易数据扩展跟性能提高。

3. 事务处理复杂。

### 第二步：其次才是水平拆分

**水平拆分的典型场景就是大家熟知的分库分表。**

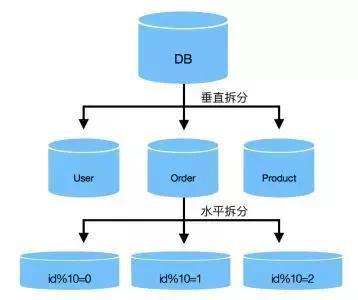
垂直拆分后遇到单机瓶颈，可以使用水平拆分。相对于垂直拆分的区别是：垂直拆分是把不同的表拆到不同的数据库中，而水平拆分是把同一个表拆到不同的数据库中。

相对于垂直拆分，水平拆分不是将表的数据做分类，而是按照某个字段的某种规则来分散到多个库之中，每个表中包含一部分数据。简单来说，我们可以将数据的水平切分理解为是按照数据行的切分，就是将表中 的某些行切分到一个数据库，而另外的某些行又切分到其他的数据库中。

分库分表需要涉及到对应的SQL路由规则主库备库等，例如：淘宝设计了一套TDDL来解决这些问题，应用端只需配置对应的规则即可，对应用端的没有任何侵入的设计。

水平拆分，总之，一般先分库，如果分库后查询仍然慢，于是按照分库的思想开始做分表的工作数据库采用分布式数据库（所有节点的数据加起来才算是整体数据），文件系统采用分布式文件系统任何强大的单一服务器都满足不了大型系统持续增长的业务需求，数据库读写分离随着业务的发展最终也将无法满足需求，需要使用分布式数据库及分布式文件系统来支撑。

### 总结，数据库拆分原则：



1.优先考虑缓存降低对数据库的读操作。

2.再考虑读写分离，降低数据库写操作。

3.最后开始数据拆分,切分模式： **首先垂直（纵向）拆分、再次水平拆分**。

4.首先考虑按照业务垂直拆分。

5.再考虑水平拆分：先分库(设置数据路由规则，把数据分配到不同的库中)

6.最后再考虑分表，单表拆分到数据1000万以内。

## MongoDB、Hbase、Redis等NoSQL优劣势、应用场景

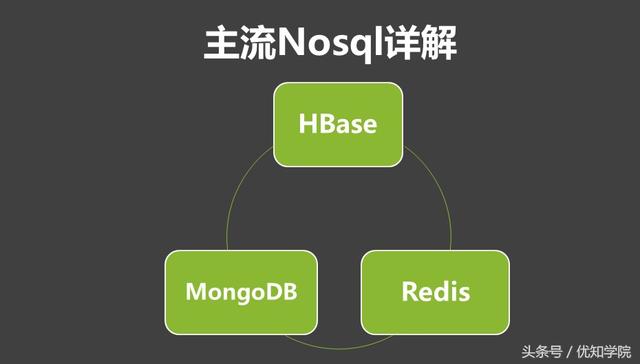
### NoSQL的四大种类

NoSQL数据库在整个数据库领域的江湖地位已经不言而喻。在大数据时代，虽然RDBMS很优秀，但是面对快速增长的数据规模和日渐复杂的数据模型，RDBMS渐渐力不从心，无法应对很多数据库处理任务，这时NoSQL凭借易扩展、大数据量和高性能以及灵活的数据模型成功的在数据库领域站稳了脚跟。

目前大家基本认同将NoSQL数据库分为四大类：键值存储数据库，文档型数据库，列存储数据库和图形数据库，其中每一种类型的数据库都能够解决关系型数据不能解决的问题。在实际应用中，NoSQL数据库的分类界限其实没有那么明显，往往会是多种类型的组合体。



### 主流nosql的详解：MongoDB、Hbase、Redis



### MongoDB

MongoDB 是一个高性能，开源，无模式的文档型数据库，开发语言是C++。它在许多场景下可用于替代统的关系型数据库或键/值存储方式。

**1.MongoDB特点**

* 所用语言：C++
* 特点：保留了SQL一些友好的特性（查询，索引）。
* 使用许可： AGPL（发起者： Apache）
* 协议： Custom, binary（ BSON）
* Master/slave复制（支持自动错误恢复，使用 sets 复制）
* 内建分片机制
* 支持 javascript表达式查询
* 可在服务器端执行任意的 javascript函数
* update-in-place支持比CouchDB更好
* 在数据存储时采用内存到文件映射
* 对性能的关注超过对功能的要求
* 建议最好打开日志功能（参数 –journal）
* 在32位操作系统上，数据库大小限制在约2.5Gb
* 空数据库大约占 192Mb
* 采用 GridFS存储大数据或元数据（不是真正的文件系统）

**2.MongoDB优点：**

1）更高的写负载，MongoDB拥有更高的插入速度。

2）处理很大的规模的单表，当数据表太大的时候可以很容易的分割表。

3）高可用性，设置M-S不仅方便而且很快，MongoDB还可以快速、安全及自动化的实现节点 （数据中心）故障转移。

4）快速的查询，MongoDB支持二维空间索引，比如管道，因此可以快速及精确的从指定位置 获取数据。MongoDB在启动后会将数据库中的数据以文件映射的方式加载到内存中。如果内 存资源相当丰富的话，这将极大地提高数据库的查询速度。

5）非结构化数据的爆发增长，增加列在有些情况下可能锁定整个数据库，或者增加负载从而 导致性能下降，由于MongoDB的弱数据结构模式，添加1个新字段不会对旧表格有任何影响， 整个过程会非常快速。

**3.MongoDB缺点：**

1）不支持事务。

2）MongoDB占用空间过大 。

3）MongoDB没有成熟的维护工具。

**4.MongoDB应用场景**

1.）适用于实时的插入、更新与查询的需求，并具备应用程序实时数据存储所需的复制及高度伸缩性；

2） 非常适合文档化格式的存储及查询；

3.）高伸缩性的场景：MongoDB 非常适合由数十或者数百台服务器组成的数据库。

4.）对性能的关注超过对功能的要求。

### HBase

HBase 是 Apache Hadoop 中的一个子项目，属于 bigtable 的开源版本，所实现的语言为Java（故依赖 Java SDK）。HBase 依托于 Hadoop 的 HDFS（分布式文件系统）作为最基本存储基础单元。

**1.HBase 特点：**

* 所用语言： Java
* 特点：支持数十亿行X上百万列
* 使用许可： Apache
* 协议：HTTP/REST （支持 Thrift，见编注4）
* 在 BigTable之后建模
* 采用分布式架构 Map/reduce
* 对实时查询进行优化
* 高性能 Thrift网关
* 通过在server端扫描及过滤实现对查询操作预判
* 支持 XML, Protobuf, 和binary的HTTP
* Cascading, hive, and pig source and sink modules
* 基于 Jruby（ JIRB）的shell
* 对配置改变和较小的升级都会重新回滚
* 不会出现单点故障
* 堪比MySQL的随机访问性能

**3. HBase 优点**

1） 存储容量大，一个表可以容纳上亿行，上百万列；

2.）可通过版本进行检索，能搜到所需的历史版本数据；

3.）负载高时，可通过简单的添加机器来实现水平切分扩展，跟Hadoop的无缝集成保障了其数据可靠性（HDFS）和海量数据分析的高性能（MapReduce）；

4.）在第3点的基础上可有效避免单点故障的发生。

**4.HBase 缺点**

1. 基于Java语言实现及Hadoop架构意味着其API更适用于Java项目；

2. node开发环境下所需依赖项较多、配置麻烦（或不知如何配置，如持久化配置），缺乏文档；

3. 占用内存很大，且鉴于建立在为批量分析而优化的HDFS上，导致读取性能不高；

4. API相比其它 NoSql 的相对笨拙。

**5.HBase 适用场景**

1）bigtable类型的数据存储；

2）对数据有版本查询需求；

3）应对超大数据量要求扩展简单的需求。

### Redis

Redis 是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。目前由VMware主持开发工作。

**1.Redis 特点：**

* 所用语言：C/C++
* 特点：运行异常快
* 使用许可： BSD
* 协议：类 Telnet
* 有硬盘存储支持的内存数据库，
* 但自2.0版本以后可以将数据交换到硬盘（注意， 2.4以后版本不支持该特性！）
* Master-slave复制（见编注3）
* 虽然采用简单数据或以键值索引的哈希表，但也支持复杂操作，例如 ZREVRANGEBYSCORE。
* INCR & co （适合计算极限值或统计数据）
* 支持 sets（同时也支持 union/diff/inter）
* 支持列表（同时也支持队列；阻塞式 pop操作）
* 支持哈希表（带有多个域的对象）
* 支持排序 sets（高得分表，适用于范围查询）
* Redis支持事务
* 支持将数据设置成过期数据（类似快速缓冲区设计）
* Pub/Sub允许用户实现消息机制

**2. Redis 优势**

1）非常丰富的数据结构；

2.）Redis提供了事务的功能，可以保证一串 命令的原子性，中间不会被任何操作打断；

3.）数据存在内存中，读写非常的高速，可以达到10w/s的频率。

**3.Redis 缺点**

1） Redis3.0后才出来官方的集群方案，但仍存在一些架构上的问题；

2.）持久化功能体验不佳——通过快照方法实现的话，需要每隔一段时间将整个数据库的数据写到磁盘上，代价非常高；而aof方法只追踪变化的数据，类似于mysql的binlog方法，但追加log可能过大，同时所有操作均要重新执行一遍，恢复速度慢；

3）由于是内存数据库，所以，单台机器，存储的数据量，跟机器本身的内存大小。虽然redis本身有key过期策略，但是还是需要提前预估和节约内存。如果内存增长过快，需要定期删除数据。

**4.Redis 应用场景：**

最佳应用场景：适用于数据变化快且数据库大小可遇见（适合内存容量）的应用程序。

例如：微博、数据分析、实时数据搜集、实时通讯等。

## 数据库分库分表、读写分离的原理实现，使用场景

### 为什么要分库分表和读写分离？

类似淘宝网这样的网站，海量数据的存储和访问成为了系统设计的瓶颈问题，日益增长的业务数据，无疑对数据库造成了相当大的负载，同时对于系统的稳定性和扩展性提出很高的要求。随着时间和业务的发展，数据库中的表会越来越多，表中的数据量也会越来越大，相应地，数据操作的开销也会越来越大；另外，无论怎样升级硬件资源，单台服务器的资源（CPU、磁盘、内存、网络IO、事务数、连接数）总是有限的，最终数据库所能承载的数据量、数据处理能力都将遭遇瓶颈。分表、分库和读写分离可以有效地减小单台数据库的压力。

### 分库分表的原理和实现

**1.什么是分区、分表、分库**

**分区**

就是把一张表的数据分成N个区块，在逻辑上看最终只是一张表，但底层是由**N个物理区块组成的，**分区实现比较简单，数据库mysql、oracle等很容易就可支持。

**分表**

就是把一张表按一定的规则分解成N个具有独立存储空间的实体表。系统读写时需要根据定义好的规则得到对应的字表明，然后操作它。

**分库**

一旦分表，一个库中的表会越来越多

将整个数据库比作图书馆，一张表就是一本书。当要在一本书中查找某项内容时，如果不**分章节**，查找的效率将会下降。而同理，在数据库中就是**分区**。

**2.什么时候考虑使用分区？**

一张表的查询速度已经慢到影响使用的时候。

* sql经过优化
* 数据量大
* 表中的数据是分段的
* 对数据的操作往往只涉及一部分数据，而不是所有的数据

**分区解决的问题**

主要可以提升查询效率

**分区的实现方式（简单），例如：**

mysql5 开始支持分区功能

CREATE TABLE sales (

id INT AUTO\_INCREMENT,

amount DOUBLE NOT NULL,

order\_day DATETIME NOT NULL,

PRIMARY KEY(id, order\_day)

) ENGINE=Innodb

PARTITION BY RANGE(YEAR(order\_day)) (

PARTITION p\_2010 VALUES LESS THAN (2010),

PARTITION p\_2011 VALUES LESS THAN (2011),

PARTITION p\_2012 VALUES LESS THAN (2012),

PARTITION p\_catchall VALUES LESS THAN MAXVALUE);

**3.什么时候考虑分表？**

* 一张表的查询速度已经慢到影响使用的时候。
* sql经过优化
* 数据量大
* 当频繁插入或者联合查询时，速度变慢

**4.分表解决的问题**

分表后，单表的并发能力提高了，磁盘I/O性能也提高了，写操作效率提高了

* 查询一次的时间短了
* 数据分布在不同的文件，磁盘I/O性能提高
* 读写锁影响的数据量变小
* 插入数据库需要重新建立索引的数据减少

**5.分表的实现方式（复杂）**

需要业务系统配合迁移升级，工作量较大。

**6.常见分表、分库常用策略：**

1.平均进行分配hash(object)%N(适用于简单架构)。

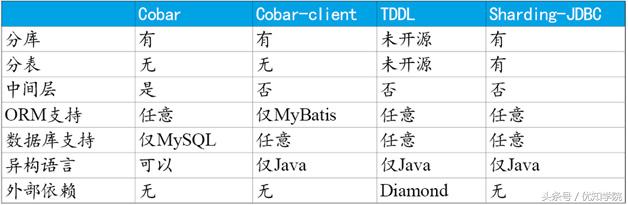
2.按照权重进行分配且均匀轮询。

3.按照业务进行分配。

4.按照一致性hash算法进行分配(适用于集群架构，在集群中节点的添加和删除不会造成数据丢失，方便数据迁移)。

**7.分库分表中间件**

分表又分为单库分表（表名不同）和多库分表（表名相同），不管使用哪种策略都还需要自己去实现路由，制定路由规则等**，**可以考虑使用开源的分库分表中间件，无侵入应用设计，例如淘宝的tddl等。



### 读写分离的原理和实现

**1、什么是读写分离**

读写分离，基本的原理是让主数据库处理事务性增、改、删操作（INSERT、UPDATE、DELETE），而从数据库处理SELECT查询操作。数据库复制被用来把事务性操作导致的变更同步到集群中的从数据库。

**2、为什么要读写分离呢？**

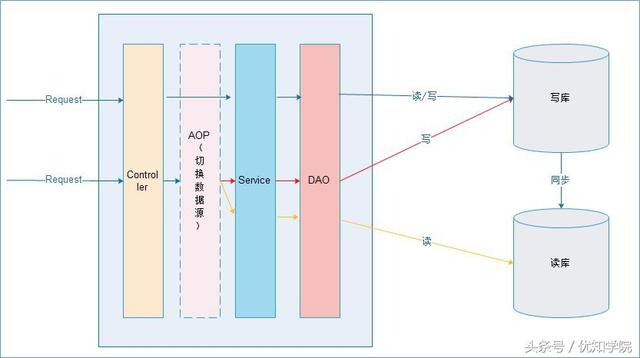
因为数据库的“写”（写10000条数据到oracle可能要3分钟）操作是比较耗时的。

但是数据库的“读”（从oracle读10000条数据可能只要5秒钟）。

**所以读写分离，解决的是，数据库的写入，影响了查询的效率。**

**3、什么时候要读写分离？**

数据库不一定要读写分离，如果程序使用数据库较多时，而更新少，查询多的情况下会考虑使用，利用数据库 主从同步 。可以减少数据库压力，提高性能。当然，数据库也有其它优化方案。memcache 或是 表折分，或是搜索引擎。都是解决方法。



**4.主从复制、读写分离的基本设计**

在实际的生产环境中，对数据库的读和写都在同一个数据库服务器中，是不能满足实际需求的。无论是在安全性、高可用性还是高并发等各个方面都是完全不能满足实际需求的。因此，通过主从复制的方式来同步数据，再通过读写分离来提升数据库的并发负载能力。

一台主、多台从，主提供写操作，从提供读操作。

读写分离的实现：

我们只需要实现读写分离，主从复制数据一般由数据库级来实现同步，当然也可以自己去实现同步，只是需要考虑的点比较多。

### 分库分表、读写分离总结：

**1.分区**

对业务透明，分区只不过把存放数据的文件分成了许多小块，根据一定的规则把数据文件(MYD)和索引文件（MYI）进行了分割，分区后的表呢，还是一张表。

**2.分表**

当数据量大到一定程度的时候，都会导致处理性能的不足，这个时候就没有办法了，只能进行分表处理。也就是把数据库当中数据根据按照分库原则分到多个数据表当中，这样，就可以把大表变成多个小表，不同的分表中数据不重复，从而提高处理效率。

**3.分库**

分表和分区都是基于同一个数据库里的数据分离技巧，对数据库性能有一定提升，但是随着业务数据量的增加，原来所有的数据都是在一个数据库上的，网络IO及文件IO都集中在一个数据库上的，因此CPU、内存、文件IO、网络IO都可能会成为系统瓶颈。

当业务系统的数据容量接近或超过单台服务器的容量、QPS/TPS接近或超过单个数据库实例的处理极限等此时，往往是采用垂直和水平结合的数据拆分方法，把数据服务和数据存储分布到多台数据库服务器上。

**4.读写分离方案**

当数据库读远大于写，查询多的情况，就可以考虑主数据负责写操作，从数据库负责读操作，一主多重，从而把数据读写分离，最后还可以结合redis等缓存来配合分担数据的读操作，大大的降低后端数据库的压力。

## 如何实现分库分表+动态数据源+读写分离

### 分布式数据层中间件：

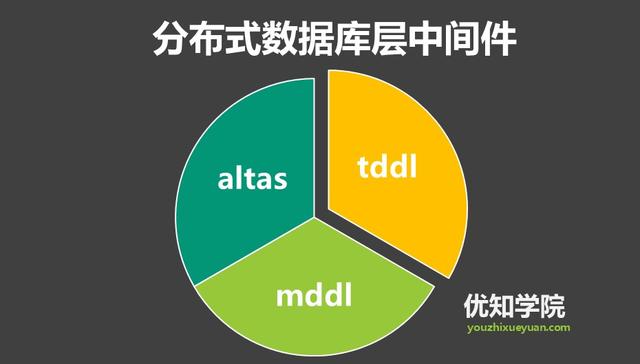
**1.简介：**

分布式数据访问层中间件，旨在为供一个通用数据访问层服务，支持MySQL动态数据源、读写分离、分布式唯一主键生成器、分库分表、动态化配置等功能，并且支持从客户端角度对数据源的各方面（比如连接池、SQL等）进行监控，后续考虑支持NoSQL、Cache等多种数据源。

**2.功能特性**

* 动态数据源
* 读写分离
* 分布式唯一主键生成器
* 分库分表
* 连接池及SQL监控
* 动态化配置

### 常见的数据层中间件：



**1.TDDL**

淘宝根据自己的业务特点开发了TDDL框架，主要解决了分库分表对应用的透明化以及异构数据库之间的数据复制，它是一个基于集中式配置的JDBC datasource实现。

**特点**

实现动态数据源、读写分离、分库分表。

**缺点**

分库分表功能还未开源，当前公布文档较少，并且需要依赖diamond（淘宝内部使用的一个管理持久配置的系统）

**2.Atlas**

Qihoo 360开发维护的一个基于MySQL协议的数据中间层项目。它实现了MySQL的客户端与服务端协议，作为服务端与应用程序通信，同时作为客户端与MySQL通信

**特点：**

实现读写分离、单库分表

**缺点：**

不支持分库分表

**3.MTDDL（Meituan Distributed Data Layer）**

美团点评分布式数据访问层中间件

**特点**

实现动态数据源、读写分离、分库分表，与tddl类似。

**下面我以MTDDL为例，也可以参考淘宝tddl，完整详解分布式数据层中间件的架构设计。**

#### 分布式数据层中间件架构设计

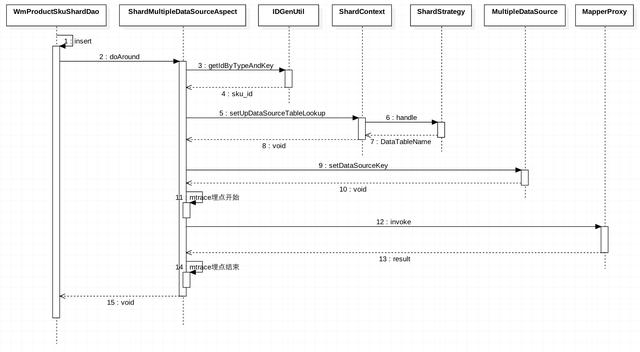
下图是一次完整的DAO层insert方法调用时序图，简单阐述了MTDDL的整个逻辑架构。

其中包含了：

**1.分布式唯一主键的获取**

**2.动态数据源的路由**

**3.以及SQL埋点监控等过程**：



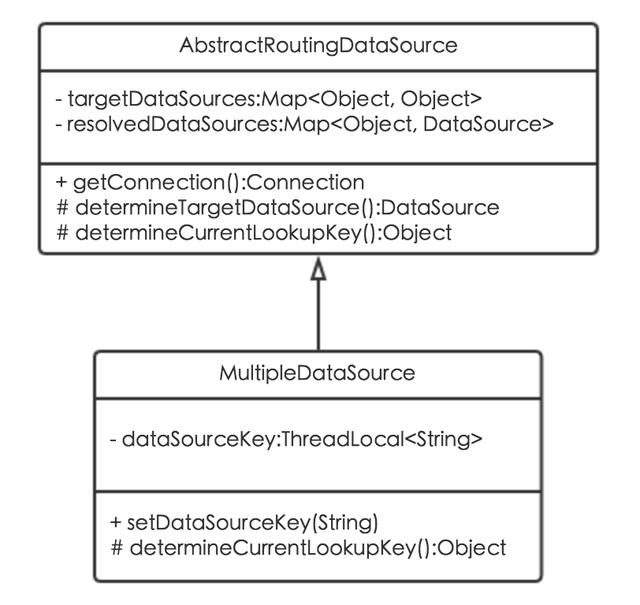
### 分布式数据层中间件：具体实现

**1.动态数据源及读写分离**

在Spring JDBC AbstractRoutingDataSource的基础上扩展出MultipleDataSource动态数据源类，通过动态数据源注解及AOP实现。

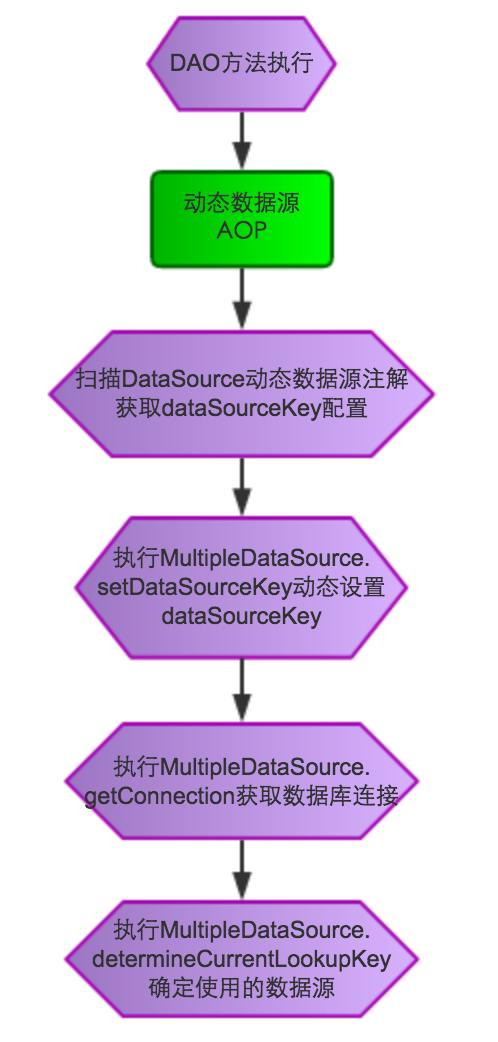
**2.动态数据源**

MultipleDataSource动态数据源类，继承于Spring JDBC AbstractRoutingDataSource抽象类，实现了determineCurrentLookupKey方法，通过setDataSourceKey方法来动态调整dataSourceKey，进而达到动态调整数据源的功能。其类图如下：



**3.动态数据源AOP**

ShardMultipleDataSourceAspect动态数据源切面类，针对DAO方法进行功能增强，通过扫描DataSource动态数据源注解来获取相应的dataSourceKey，从而指定具体的数据源。具体流程图如下：



**4.配置和使用方式举例**

/\*\*

\* 参考配置

\*/

<bean id="multipleDataSource" class="com.sankuai.meituan.waimai.datasource.multi.MultipleDataSource">

/\*\* 数据源配置 \*/

<property name="targetDataSources">

<map key-type="java.lang.String">

/\*\* 写数据源 \*/

<entry key="dbProductWrite" value-ref="dbProductWrite"/>

/\*\* 读数据源 \*/

<entry key="dbProductRead" value-ref="dbProductRead"/>

</map>

</property>

</bean>

/\*\*

\* DAO使用动态数据源注解

\*/

public interface WmProductSkuDao {

/\*\* 增删改走写数据源 \*/

@DataSource("dbProductWrite")

public void insert(WmProductSku sku);

/\*\* 查询走读数据源 \*/

@DataSource("dbProductRead")

public void getById(long sku\_id);

}

**5.分布式唯一主键生成器**

**众所周知，分库分表首先要解决的就是分布式唯一主键的问题**，业界也有很多相关方案：

序号实现方案优点缺点UUID本地生成，不需要RPC，低延时；

扩展性好，基本没有性能上限无法保证趋势递增；

UUID过长128位，不易存储，往往用字符串表示2Snowflake或MongoDB ObjectId分布式生成，无单点；

趋势递增，生成效率快没有全局时钟的情况下，只能保证趋势递增；

当通过NTP进行时钟同步时可能会出现重复ID；

数据间隙较大3proxy服务+数据库分段获取ID分布式生成，段用完后需要去DB获取，同server有序可能产生数据空洞，即有些ID没有分配就被跳过了，主要原因是在服务重启的时候发生；

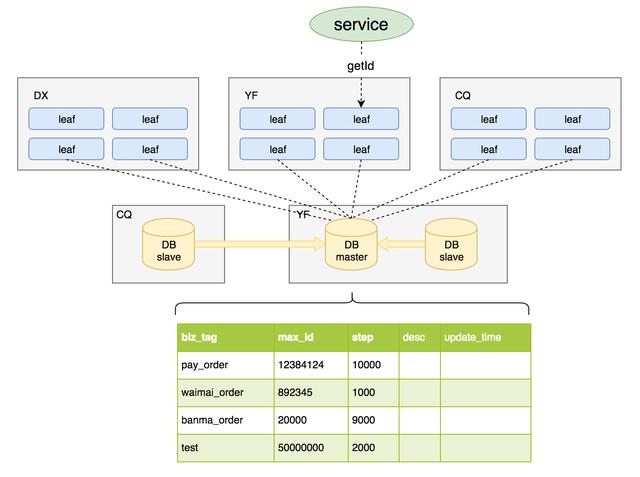
无法保证有序，需要未来解决，可能会通过其他接口方案实现

综上，方案3的缺点可以通过一些手段避免，但其他方案的缺点不好处理，所以选择第3种方案：分布式ID生成系统Leaf。

**6.分布式ID生成系统Leaf**

分布式ID生成系统Leaf，其实是一种基于DB的Ticket服务，通过一张通用的Ticket表来实现分布式ID的持久化，执行update更新语句来获取一批Ticket，这些获取到的Ticket会在内存中进行分配，分配完之后再从DB获取下一批Ticket。

**整体架构图如下：**

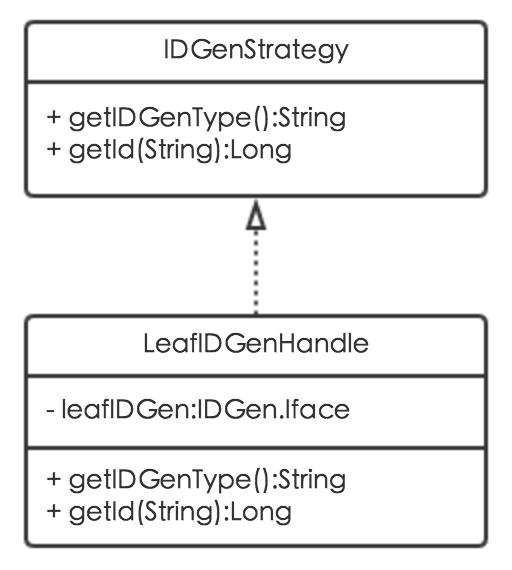


每个业务tag对应一条DB记录，DB MaxID字段记录当前该Tag已分配出去的最大ID值。

IDGenerator服务启动之初向DB申请一个号段，传入号段长度如 genStep = 10000，DB事务置 MaxID = MaxID + genStep，DB设置成功代表号段分配成功。每次IDGenerator号段分配都通过原子加的方式，待分配完毕后重新申请新号段。

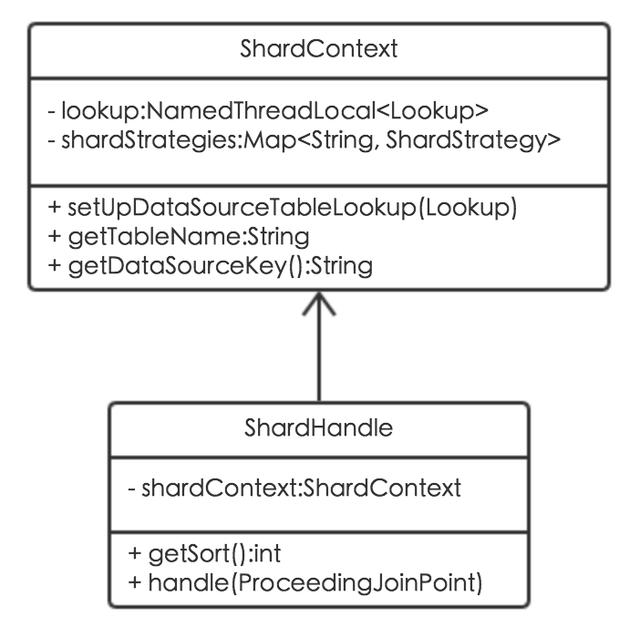
**7.唯一主键生成算法扩展**

MTDDL不仅集成了Leaf算法，还支持唯一主键算法的扩展，通过新增唯一主键生成策略类实现IDGenStrategy接口即可。IDGenStrategy接口包含两个方法：getIDGenType用来指定唯一主键生成策略，getId用来实现具体的唯一主键生成算法。其类图如下：

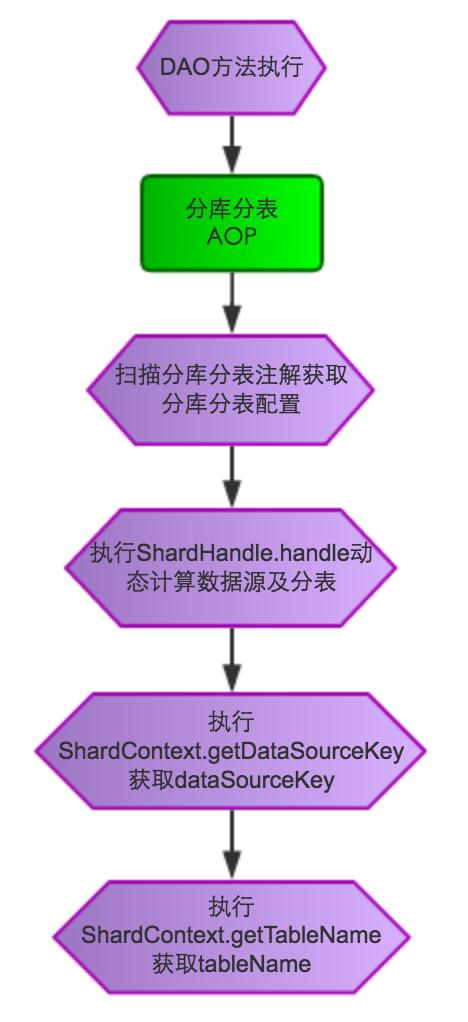


**8.分库分表**

在动态数据源AOP的基础上扩展出分库分表AOP，通过分库分表ShardHandle类实现分库分表数据源路由及分表计算。ShardHandle关联了分库分表上下文ShardContext类，而ShardContext封装了所有的分库分表算法。其类图如下：



**分库分表流程图如下：**



**9.分库分表取模算法**

分库分表目前默认使用的是取模算法，分表算法为 (#shard\_key % (group\_shard\_num \* table\_shard\_num))，分库算法为 (#shard\_key % (group\_shard\_num \* table\_shard\_num)) / table\_shard\_num，其中group\_shard\_num为分库个数，table\_shard\_num为每个库的分表个数。

例如把一张大表分成100张小表然后散到2个库，则0-49落在第一个库、50-99落在第二个库。核心实现如下：

public class ModStrategyHandle implements ShardStrategy {

@Override

public String getShardType() {

return "mod";

}

@Override

public DataTableName handle(String tableName, String dataSourceKey, int tableShardNum,

int dbShardNum, Object shardValue) {

/\*\* 计算散到表的值 \*/

long shard\_value = Long.valueOf(shardValue.toString());

long tablePosition = shard\_value % tableShardNum;

long dbPosition = tablePosition / (tableShardNum / dbShardNum);

String finalTableName = new StringBuilder().append(tableName).append("\_").append(tablePosition).toString();

String finalDataSourceKey = new StringBuilder().append(dataSourceKey).append(dbPosition).toString();

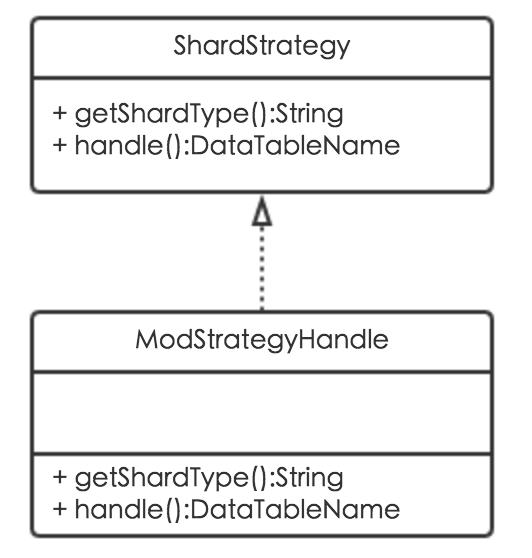
return new DataTableName(finalTableName, finalDataSourceKey);

}

}

**10.分库分表算法扩展**

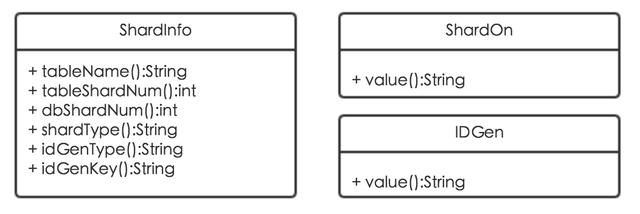
MTDDL不仅支持分库分表取模算法，还支持分库分表算法的扩展，通过新增分库分表策略类实现ShardStrategy接口即可。ShardStrategy接口包含两个方法：getShardType用来指定分库分表策略，handle用来实现具体的数据源及分表计算逻辑。其类图如下：



**11.全注解方式接入**

为了尽可能地方便业务方接入，MTDDL采用全注解方式使用分库分表功能，通过ShardInfo、ShardOn、IDGen三个注解实现。

ShardInfo注解用来指定具体的分库分表配置：包括分表名前缀tableName、分表数量tableShardNum、分库数量dbShardNum、分库分表策略shardType、唯一键生成策略idGenType、唯一键业务方标识idGenKey；ShardOn注解用来指定分库分表字段；IDGen注解用来指定唯一键字段。具体类图如下：



**12.配置和使用方式举例**

// 动态数据源

@DataSource("dbProductSku")

// tableName：分表名前缀，tableShardNum：分表数量，dbShardNum：分库数量，shardType：分库分表策略，idGenType：唯一键生成策略，idGenKey：唯一键业务方标识

@ShardInfo(tableName="wm\_food", tableShardNum=100, dbShardNum=1, shardType="mod", idGenType=IDGenType.LEAF, idGenKey=LeafKey.SKU)

@Component

public interface WmProductSkuShardDao {

// @ShardOn("wm\_poi\_id") 将该注解修饰的对象的wm\_poi\_id字段作为shardValue

// @IDGen("id") 指定要设置唯一键的字段

public void insert(@ShardOn("wm\_poi\_id") @IDGen("id") WmProductSku sku);

// @ShardOn 将该注解修饰的参数作为shardValue

public List<WmProductSku> getSkusByWmPoiId(@ShardOn long wm\_poi\_id);

}

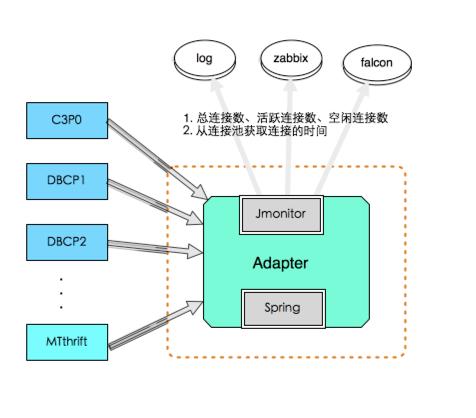
### 连接池及SQL监控

DB连接池使用不合理容易引发很多问题，如连接池最大连接数设置过小导致线程获取不到连接、获取连接等待时间设置过大导致很多线程挂起、空闲连接回收器运行周期过长导致空闲连接回收不及时等等，如果缺乏有效准确的监控，会造成无法快速定位问题以及追溯历史。

**连接池监控**

**实现方案**

结合Spring完美适配c3p0、dbcp1、dbcp2、mtthrift等多种方案，自动发现新加入到Spring容器中的数据源进行监控，通过美团点评统一监控组件JMonitor上报监控数据。整体架构图如下：



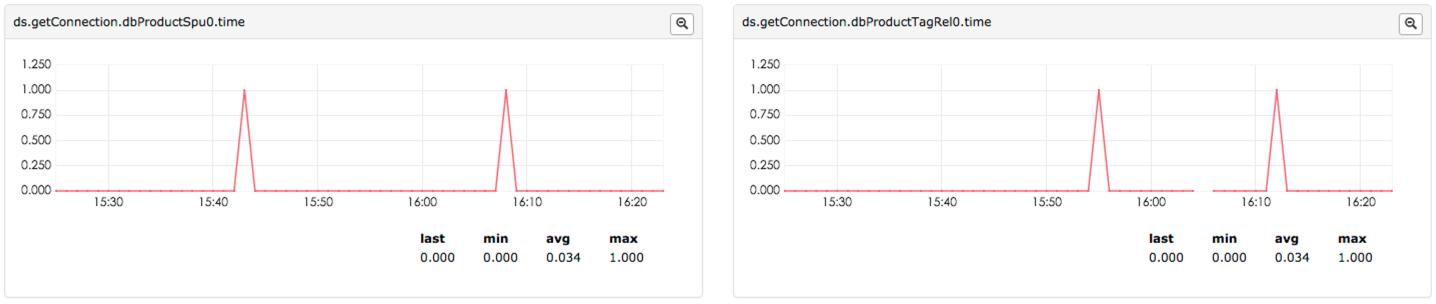
**连接数量监控**

监控连接池active、idle、total连接数量，Counter格式：（连接池类型.数据源.active/idle/total\_connection），效果图如下：



**获取连接时间监控**

监控获取空闲连接时间，Counter格式：（ds.getConnection.数据源.time），效果图如下：



# ****BAT必考JVM系列专题****

## JVM性能调优的6大步骤，及关键调优参数详解

### JVM内存调优

jvm性能挑优详解

jvm调优工具，内存泄漏，挑优参数，调优步骤

**对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数。**

**1.Full GC**

会对整个堆进行整理，包括Young、Tenured和Perm。Full GC因为需要对整个堆进行回收，所以比较慢，因此应该尽可能减少Full GC的次数。

**2.导致Full GC的原因**

1)年老代（Tenured）被写满

调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象 。

**2)持久代Pemanet Generation空间不足**

增大Perm Gen空间，避免太多静态对象 ， 控制好新生代和旧生代的比例

**3)System.gc()被显示调用**

垃圾回收不要手动触发，尽量依靠JVM自身的机制

**在对JVM调优的过程中，很大一部分工作就是对于FullGC的调节，下面详细介绍对应JVM调优的方法和步骤。**

### JVM性能调优方法和步骤

**调整jvm参数、监控gc状态、分析内存泄露、分析dump文件**

**1.监控GC的状态**

使用各种JVM工具，查看当前日志，分析当前JVM参数设置，并且分析当前堆内存快照和gc日志，根据实际的各区域内存划分和GC执行时间，觉得是否进行优化。

**举一个例子： 系统崩溃前的一些现象：**

* 每次垃圾回收的时间越来越长，由之前的10ms延长到50ms左右，FullGC的时间也有之前的0.5s延长到4、5s
* FullGC的次数越来越多，最频繁时隔不到1分钟就进行一次FullGC
* 年老代的内存越来越大并且每次FullGC后年老代没有内存被释放

之后系统会无法响应新的请求，逐渐到达OutOfMemoryError的临界值，这个时候就需要分析JVM内存快照dump。

**2.生成堆的dump文件**

通过JMX的MBean生成当前的Heap信息，大小为一个3G（整个堆的大小）的hprof文件，如果没有启动JMX可以通过Java的jmap命令来生成该文件。

**3.分析dump文件**

打开这个3G的堆信息文件，显然一般的Window系统没有这么大的内存，必须借助高配置的Linux，几种工具打开该文件：

* Visual VM
* IBM HeapAnalyzer
* JDK 自带的Hprof工具

**Mat(Eclipse专门的静态内存分析工具)推荐使用**

备注：文件太大，建议使用Eclipse专门的静态内存分析工具Mat打开分析。

**4.分析结果，判断是否需要优化**

如果各项参数设置合理，系统没有超时日志出现，GC频率不高，GC耗时不高，那么没有必要进行GC优化，如果GC时间超过1-3秒，或者频繁GC，则必须优化。

**注：如果满足下面的指标，则一般不需要进行GC：**

* Minor GC执行时间不到50ms；
* Minor GC执行不频繁，约10秒一次；
* Full GC执行时间不到1s；
* Full GC执行频率不算频繁，不低于10分钟1次；

**5.调整GC类型和内存分配**

如果内存分配过大或过小，或者采用的GC收集器比较慢，则应该优先调整这些参数，并且先找1台或几台机器进行beta，然后比较优化过的机器和没有优化的机器的性能对比，并有针对性的做出最后选择。

**6.不断的分析和调整**

通过不断的试验和试错，分析并找到最合适的参数，如果找到了最合适的参数，则将这些参数应用到所有服务器。



cms参数优化步流程

**下面我再继续介绍下JVM的关键参数配置(仅用于参考)。**

#### JVM调优参数参考

1.针对JVM堆的设置，一般可以通过-Xms -Xmx限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，通常把最大、最小设置为相同的值;

2.年轻代和年老代将根据默认的比例（1：2）分配堆内存， 可以通过调整二者之间的比率NewRadio来调整二者之间的大小，也可以针对回收代。

比如年轻代，通过 -XX:newSize -XX:MaxNewSize来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX:newSize -XX:MaxNewSize设置为同样大小。

3.年轻代和年老代设置多大才算合理

**1）更大的年轻代必然导致更小的年老代，大的年轻代会延长普通GC的周期，但会增加每次GC的时间；小的年老代会导致更频繁的Full GC**

**2）更小的年轻代必然导致更大年老代，小的年轻代会导致普通GC很频繁，但每次的GC时间会更短；大的年老代会减少Full GC的频率**

如何选择应该依赖应用程序对象生命周期的分布情况： 如果应用存在大量的临时对象，应该选择更大的年轻代；如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性。

**在抉择时应该根 据以下两点：**

（1）本着Full GC尽量少的原则，让年老代尽量缓存常用对象，JVM的默认比例1：2也是这个道理 。

（2）通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响Full GC的前提下，根据实际情况加大年轻代，比如可以把比例控制在1：1。但应该给年老代至少预留1/3的增长空间。

4.在配置较好的机器上（比如多核、大内存），可以为年老代选择并行收集算法： -XX:+UseParallelOldGC 。

5.线程堆栈的设置：每个线程默认会开启1M的堆栈，用于存放栈帧、调用参数、局部变量等，对大多数应用而言这个默认值太了，一般256K就足用。

理论上，在内存不变的情况下，减少每个线程的堆栈，可以产生更多的线程，但这实际上还受限于操作系统。

## 深入详解JVM内存模型与JVM参数详细配置

**1.JVM内存模型**

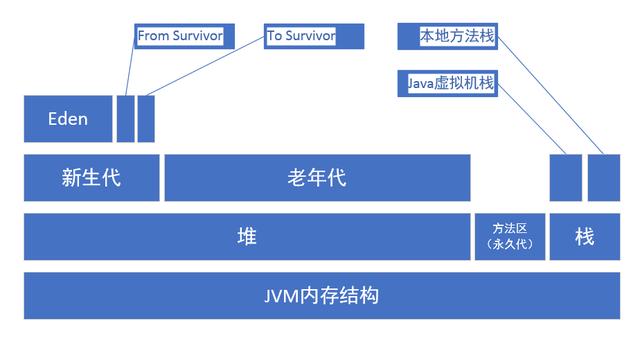
2.JVM垃圾回收算法

3.JVM垃圾回收器

4.JVM参数详解

5.JVM性能调优

### JVM内存结构



由上图可以清楚的看到JVM的内存空间分为3大部分：

1. **堆内存**
2. **方法区**
3. **栈内存**

其中栈内存可以再细分为java虚拟机栈和本地方法栈,堆内存可以划分为新生代和老年代,新生代中还可以再次划分为Eden区、From Survivor区和To Survivor区。

其中一部分是线程共享的，包括 Java 堆和方法区；另一部分是线程私有的，包括虚拟机栈和本地方法栈，以及程序计数器这一小部分内存。

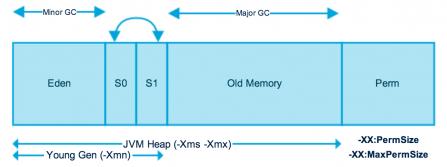
### 堆内存（Heap）

java 堆（Java Heap）是Java 虚拟机所管理的内存中最大的一块。堆是被所有**线程共享**的区域，实在虚拟机启动时创建的。堆里面存放的都是**对象的实例**（new 出来的对象都存在堆中）。

**此内存区域的唯一目的就是存放对象实例（new的对象），几乎所有的对象实例都在这里分配内存。**

堆内存分为两个部分：年轻代和老年代。我们平常所说的垃圾回收，主要回收的就是堆区。更细一点划分新生代又可划分为Eden区和2个Survivor区（From Survivor和To Survivor）。

下图中的Perm代表的是永久代，但是注意永久代并不属于堆内存中的一部分，同时jdk1.8之后永久代已经被移除。



新生代 ( Young ) 与老年代 ( Old ) 的比例的值为 1:2 ( 该值可以通过参数 –XX:NewRatio 来指定 )

默认的，Eden : from : to = 8 : 1 : 1 ( 可以通过参数 –XX:SurvivorRatio 来设定 )，即： Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。

#### 方法区（Method Area）

方法区也称”永久代“，它用于存储虚拟机加载的类信息、常量、静态变量、是各个线程共享的内存区域。

在JDK8之前的HotSpot JVM，存放这些”永久的”的区域叫做“永久代(permanent generation)”。永久代是一片连续的堆空间，在JVM启动之前通过在命令行设置参数-XX:MaxPermSize来设定永久代最大可分配的内存空间，默认大小是64M（64位JVM默认是85M）。

随着JDK8的到来，JVM不再有 永久代(PermGen)。但类的元数据信息（metadata）还在，只不过不再是存储在连续的堆空间上，而是移动到叫做“Metaspace”的本地内存（Native memory。

**方法区或永生代相关设置**

 -XX:PermSize=64MB 最小尺寸，初始分配

 -XX:MaxPermSize=256MB 最大允许分配尺寸，按需分配

 XX:+CMSClassUnloadingEnabled -XX:+CMSPermGenSweepingEnabled 设置垃圾不回收

 默认大小

 -server选项下默认MaxPermSize为64m

 -client选项下默认MaxPermSize为32m

#### 虚拟机栈(JVM Stack)

java虚拟机栈是线程私有，生命周期与线程相同。创建线程的时候就会创建一个java虚拟机栈。

虚拟机执行java程序的时候，每个方法都会创建一个栈帧，栈帧存放在java虚拟机栈中，通过压栈出栈的方式进行方法调用。

栈帧又分为一下几个区域：局部变量表、操作数栈、动态连接、方法出口等。  
平时我们所说的变量存在栈中，这句话说的不太严谨，应该说局部变量存放在java虚拟机栈的局部变量表中。  
java的8中基本类型的局部变量的值存放在虚拟机栈的局部变量表中，如果是引用型的变量，则只存储对象的引用地址。

#### 本地方法栈(Native Stack)

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

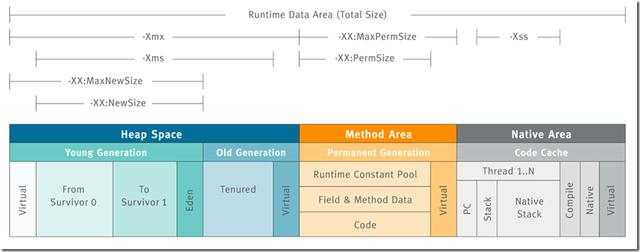
#### 程序计数器（PC Register）

程序计数器就是记录当前线程执行程序的位置，改变计数器的值来确定执行的下一条指令，比如循环、分支、方法跳转、异常处理，线程恢复都是依赖程序计数器来完成。  
Java虚拟机多线程是通过线程轮流切换并分配处理器执行时间的方式实现的。为了线程切换能恢复到正确的位置，每条线程都需要一个独立的程序计数器，所以它是线程私有的。

#### 直接内存

直接内存并不是虚拟机内存的一部分，也不是Java虚拟机规范中定义的内存区域。jdk1.4中新加入的NIO，引入了通道与缓冲区的IO方式，它可以调用Native方法直接分配堆外内存，这个堆外内存就是本机内存，不会影响到堆内存的大小。

#### **JVM内存参数设置**



 -Xms设置堆的最小空间大小。

 -Xmx设置堆的最大空间大小。

 -Xmn:设置年轻代大小

 -XX:NewSize设置新生代最小空间大小。

 -XX:MaxNewSize设置新生代最大空间大小。

 -XX:PermSize设置永久代最小空间大小。

 -XX:MaxPermSize设置永久代最大空间大小。

 -Xss设置每个线程的堆栈大小

 -XX:+UseParallelGC:选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下,年轻代使用并发收集,而年老代仍旧使用串行收集。

 -XX:ParallelGCThreads=20:配置并行收集器的线程数,即:同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

#### **典型JVM参数配置参考:**

 java-Xmx3550m-Xms3550m-Xmn2g-Xss128k

 -XX:ParallelGCThreads=20

 -XX:+UseConcMarkSweepGC-XX:+UseParNewGC

-Xmx3550m:设置JVM最大可用内存为3550M。

-Xms3550m:设置JVM促使内存为3550m。此值可以设置与-Xmx相同,以避免每次垃圾回收完成后JVM重新分配内存。

-Xmn2g:设置年轻代大小为2G。整个堆大小=年轻代大小+年老代大小+持久代大小。持久代一般固定大小为64m,所以增大年轻代后,将会减小年老代大小。此值对系统性能影响较大,官方推荐配置为整个堆的3/8。

-Xss128k:设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K。更具应用的线程所需内存大  
小进行调整。在相同物理内存下,减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在3000~5000  
左右。

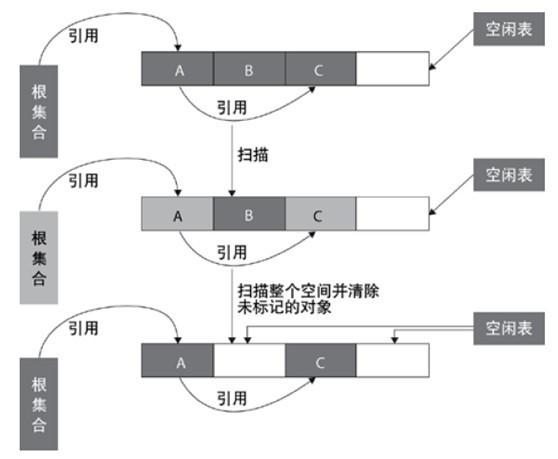
## JVM的4种垃圾回收算法、垃圾回收机制与总结

### **垃圾回收算法**

**1.标记清除**

标记-清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。

在标记阶段首先通过根节点(GC Roots)，标记所有从根节点开始的对象，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。



适用场合：

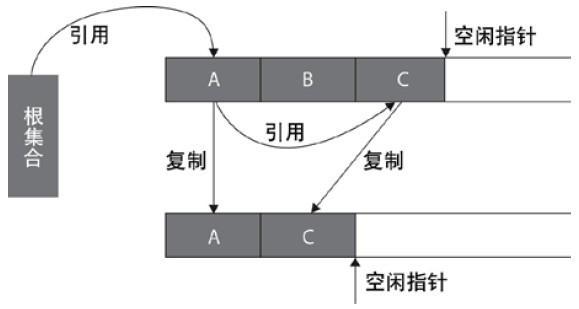
* 存活对象较多的情况下比较高效
* 适用于年老代（即旧生代）

缺点：

* 容易产生内存碎片，再来一个比较大的对象时（典型情况：该对象的大小大于空闲表中的每一块儿大小但是小于其中两块儿的和），会提前触发垃圾回收
* 扫描了整个空间两次（第一次：标记存活对象；第二次：清除没有标记的对象）

**2.复制算法**

从根集合节点进行扫描，标记出所有的存活对象，并将这些存活的对象复制到一块儿新的内存（图中下边的那一块儿内存）上去，之后将原来的那一块儿内存（图中上边的那一块儿内存）全部回收掉



**现在的商业虚拟机都采用这种收集算法来回收新生代。**

**适用场合：**

* 存活对象较少的情况下比较高效
* 扫描了整个空间一次（标记存活对象并复制移动）
* 适用于年轻代（即新生代）：基本上98%的对象是”朝生夕死”的，存活下来的会很少

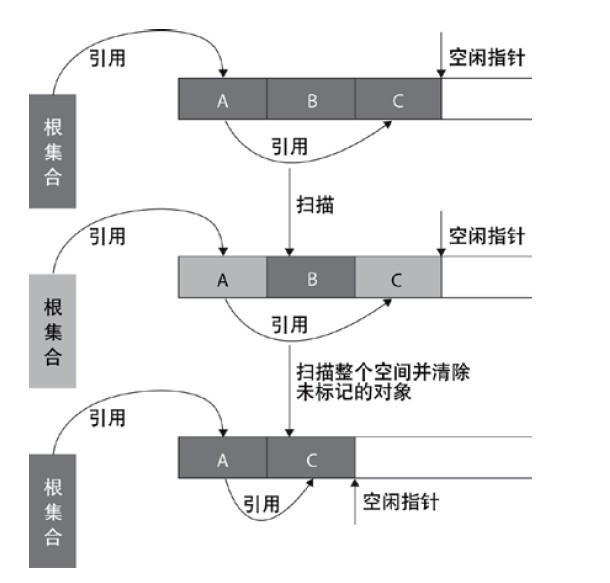
**缺点：**

* 需要一块儿空的内存空间
* 需要复制移动对象

**3.标记整理**

复制算法的高效性是建立在存活对象少、垃圾对象多的前提下的。

这种情况在新生代经常发生，但是在老年代更常见的情况是大部分对象都是存活对象。如果依然使用复制算法，由于存活的对象较多，复制的成本也将很高。



标记-压缩算法是一种老年代的回收算法，它在标记-清除算法的基础上做了一些优化。

首先也需要从根节点开始对所有可达对象做一次标记，但之后，它并不简单地清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。

**4.分代收集算法**

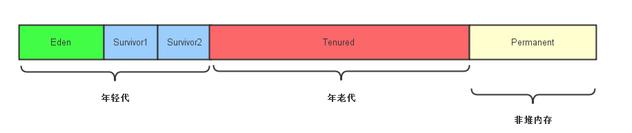
分代收集算法就是目前虚拟机使用的回收算法，它解决了标记整理不适用于老年代的问题，将内存分为各个年代。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），在堆区之外还有一个代就是永久代（Permanet Generation）。

在不同年代使用不同的算法，从而使用最合适的算法，新生代存活率低，可以使用复制算法。而老年代对象存活率搞，没有额外空间对它进行分配担保，所以只能使用标记清除或者标记整理算法。



### 垃圾回收机制

年轻代分为Eden区和survivor区（两块儿：from和to），且Eden:from:to==8:1:1。



jvm内存结构

1）新产生的对象优先分配在Eden区（除非配置了-XX:PretenureSizeThreshold，大于该值的对象会直接进入年老代）；

2）当Eden区满了或放不下了，这时候其中存活的对象会复制到from区。

这里，需要注意的是，如果存活下来的对象from区都放不下，则这些存活下来的对象全部进入年老代。之后Eden区的内存全部回收掉。

3）之后产生的对象继续分配在Eden区，当Eden区又满了或放不下了，这时候将会把Eden区和from区存活下来的对象复制到to区（同理，如果存活下来的对象to区都放不下，则这些存活下来的对象全部进入年老代），之后回收掉Eden区和from区的所有内存。

4）如上这样，会有很多对象会被复制很多次（每复制一次，对象的年龄就+1），默认情况下，当对象被复制了15次（这个次数可以通过：-XX:MaxTenuringThreshold来配置），就会进入年老代了。

5）当年老代满了或者存放不下将要进入年老代的存活对象的时候，就会发生一次Full GC（这个是我们最需要减少的，因为耗时很严重）。

### 垃圾回收有两种类型：Minor GC 和 Full GC。

**1.Minor GC**

对新生代进行回收，不会影响到年老代。因为新生代的 Java 对象大多死亡频繁，所以 Minor GC 非常频繁，一般在这里使用速度快、效率高的算法，使垃圾回收能尽快完成。

**2.Full GC**

也叫  
Major GC，对整个堆进行回收，包括新生代和老年代。由于Full GC需要对整个堆进行回收，所以比Minor  
GC要慢，因此应该尽可能减少Full GC的次数，导致Full  
GC的原因包括：老年代被写满、永久代（Perm）被写满和System.gc()被显式调用等。

### 垃圾回收算法总结

**1.年轻代：复制算法**

1) 所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

2)  
新生代内存按照8:1:1的比例分为一个eden区和两个survivor(survivor0,survivor1)区。一个Eden区，两个  
Survivor区(一般而言)。大部分对象在Eden区中生成。回收时先将eden区存活对象复制到一个survivor0区，然后清空eden区，当这个survivor0区也存放满了时，则将eden区和survivor0区存活对象复制到另一个survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后将survivor0区和survivor1区交换，即保持survivor1区为空，  
如此往复。

3) 当survivor1区不足以存放 eden和survivor0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC(Major GC)，也就是新生代、老年代都进行回收。

4) 新生代发生的GC也叫做Minor GC，MinorGC发生频率比较高(不一定等Eden区满了才触发)。

**2.年老代：标记-清除或标记-整理**

1) 在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

2) 内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。

以上这种年轻代与年老代分别采用不同回收算法的方式称为”分代收集算法”，这也是当下企业使用的一种方式

3. 每一种算法都会有很多不同的垃圾回收器去实现，在实际使用中，根据自己的业务特点做出选择就好。

## [**7种JVM垃圾收集器特点，优劣势、及使用场景**](http://youzhixueyuan.com/jvm-garbage-collector.html)

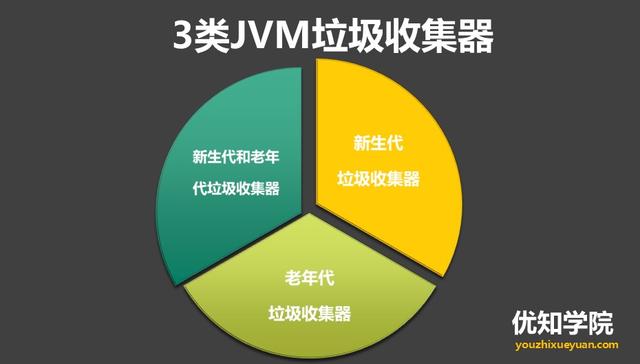
### 直通BAT之JVM系列

[直通BAT必考题系列：JVM的4种垃圾回收算法、垃圾回收机制与总结](http://youzhixueyuan.com/jvm-garbage-collection-algorithm.html)

[直通BAT必考题系列：深入详解JVM内存模型与JVM参数详细配置](http://youzhixueyuan.com/jvm-memory-model-and-parameter-configuration.html)

今天继续JVM的垃圾回收器详解，如果说垃圾收集算法是JVM内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

### 常见的垃圾收集器有3类：



**1.新生代的收集器包括**：

1. Serial
2. PraNew
3. Parallel Scavenge

**2.老年代的收集器包括：**

1. Serial Old
2. Parallel Old
3. CMS

**3.回收整个Java堆(新生代和老年代)**

1. G1收集器

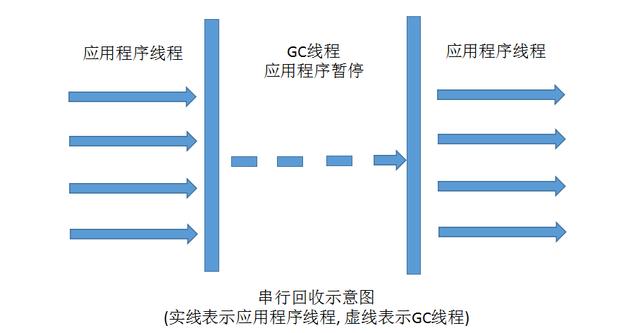
**今天我们详细谈谈以上7种垃圾收集器的优劣势和使用场景。**



### **新生代垃圾收集器**

**1.Serial串行收集器-复制算法**

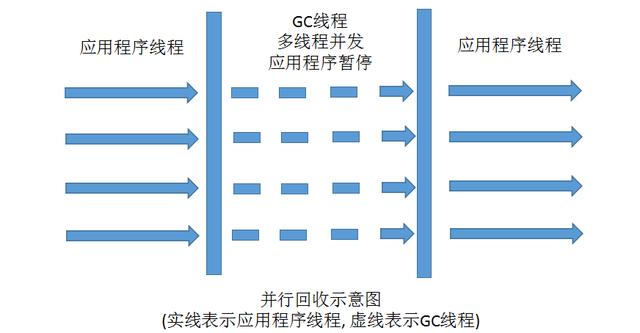
Serial收集器是新生代单线程收集器，优点是简单高效，算是最基本、发展历史最悠久的收集器。它在进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集完成。



Serial收集器依然是虚拟机运行在Client模式下默认新生代收集器，对于运行在Client模式下的虚拟机来说是一个很好的选择。

**2.ParNew收集器-复制算法**

ParNew收集器是**新生代并行收集器**，其实就是Serial收集器的多线程版本。



除了使用多线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop The Worl、对象分配规则、回收策略等都与Serial 收集器完全一样。

**3.Parallel Scavenge（并行回收）收集器-复制算法**

Parallel Scavenge收集器是新生代并行收集器，追求高吞吐量，高效利用 CPU。

该收集器的目标是达到一个可控制的吞吐量（Throughput）。所谓吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即 吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可用高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

### 老年代垃圾收集器

**1.Serial Old 收集器-标记整理算法**

Serial Old是Serial收集器的老年代版本，它同样是一个单线程(串行)收集器，使用标记整理算法。这个收集器的主要意义也是在于**给Client模式下的虚拟机使用**。

**如果在Server模式下，主要两大用途：**

（1）在JDK1.5以及之前的版本中与Parallel Scavenge收集器搭配使用

（2）作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用

**2.Parallel Old 收集器-标记整理算法**

Parallel Old 是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器在1.6中才开始提供。

**3.CMS收集器-标记整理算法**

CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为目标的收集器。

目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务器的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。

**CMS收集器是基于“标记-清除”算法实现的，**它的运作过程相对前面几种收集器来说更复杂一些，整个过程分为4个步骤：

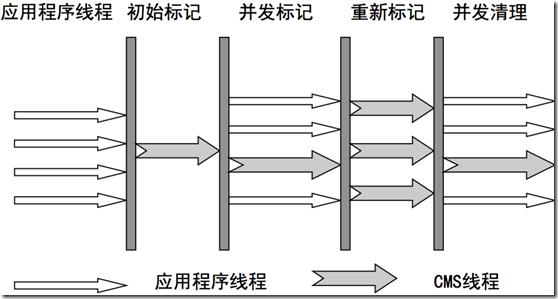
（1）初始标记

（2）并发标记

（3）重新标记

（4）并发清除

其中，初始标记、重新标记这两个步骤仍然需要“Stop The World”



**CMS收集器主要优点**：

1. 并发收集
2. 低停顿

**CMS三个明显的缺点：**

（1）CMS收集器对CPU资源非常敏感。CPU个数少于4个时，CMS对于用户程序的影响就可能变得很大，为了应付这种情况，虚拟机提供了一种称为“增量式并发收集器”的CMS收集器变种。

（2）CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。在JDK1.5的默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活。

（3）CMS是基于“标记-清除”算法实现的收集器，手机结束时会有大量空间碎片产生。空间碎片过多，可能会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前出发FullGC。

### 新生代和老年代垃圾收集器

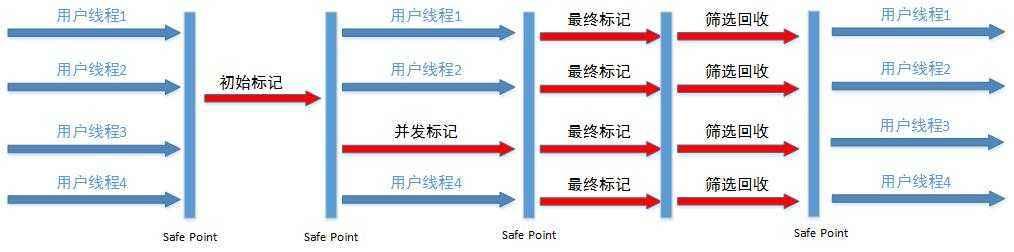
**1.G1收集器-标记整理算法**

**JDK1.7后全新的回收器, 用于取代CMS收集器。**

**G1收集器的优势：**

1. 独特的分代垃圾回收器,分代GC: 分代收集器, 同时兼顾年轻代和老年代
2. 使用分区算法, 不要求eden, 年轻代或老年代的空间都连续
3. 并行性: 回收期间, 可由多个线程同时工作, 有效利用多核cpu资源
4. 空间整理: 回收过程中, 会进行适当对象移动, 减少空间碎片
5. 可预见性: G1可选取部分区域进行回收, 可以缩小回收范围, 减少全局停顿

**G1收集器的运作大致可划分为一下步骤：**



**G1收集器的阶段分以下几个步骤：**

1、初始标记（它标记了从GC Root开始直接可达的对象）

2、并发标记（从GC Roots开始对堆中对象进行可达性分析，找出存活对象）

3、最终标记（标记那些在并发标记阶段发生变化的对象，将被回收）

4、筛选回收（首先对各个Regin的回收价值和成本进行排序，根据用户所期待的GC停顿时间指定回收计划，回收一部分Region）

### JVM垃圾收集器总结

本文主要介绍了JVM中的垃圾回收器，主要包括串行回收器、并行回收器以及CMS回收器、G1回收器。他们各自都有优缺点，通常来说你需要根据你的业务，进行基于垃圾回收器的性能测试，然后再做选择。下面给出配置回收器时，经常使用的参数：

-XX:+UseSerialGC：在新生代和老年代使用串行收集器

-XX:+UseParNewGC：在新生代使用并行收集器

-XX:+UseParallelGC ：新生代使用并行回收收集器，更加关注吞吐量

-XX:+UseParallelOldGC：老年代使用并行回收收集器

-XX:ParallelGCThreads：设置用于垃圾回收的线程数

-XX:+UseConcMarkSweepGC：新生代使用并行收集器，老年代使用CMS+串行收集器

-XX:ParallelCMSThreads：设定CMS的线程数量

-XX:+UseG1GC：启用G1垃圾回收器

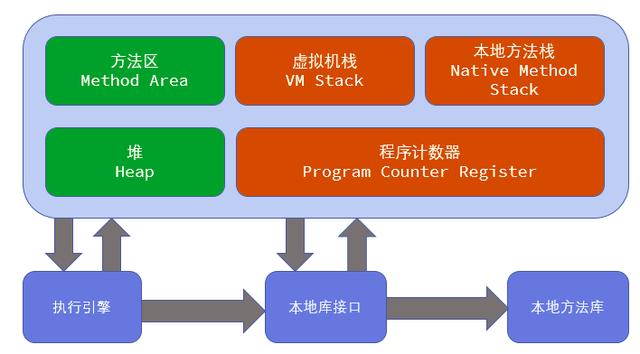
## [**JVM的内存分配、运行原理、回收算法机制**](http://youzhixueyuan.com/memory-model-and-recovery-algorithm-mechanism-of-jvm.html)

memory-model-and-recovery-algorithm-mechanism-of-jvm.html

不管是BAT面试，还是工作实践中的JVM调优以及参数设置，或者内存溢出检测等，都需要涉及到Java虚拟机的内存模型、内存分配，以及回收算法机制等，这些都是必考、必会技能。

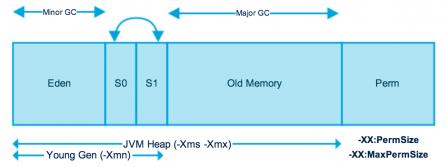
### JVM内存模型

JVM内存模型可以分为两个部分，如下图所示，**堆和方法区是所有线程共有的**，**而虚拟机栈，本地方法栈和程序计数器则是线程私有的。**



###### 1. 堆（Heap）

堆内存是所有线程共有的，可以分为两个部分：年轻代和老年代。下图中的Perm代表的是永久代，但是注意永久代并不属于堆内存中的一部分，**同时jdk1.8之后永久代也将被移除**。



堆是java虚拟机所管理的内存中**最大的**一块内存区域，也是被各个**线程共享**的内存区域，该内存区域**存放了对象实例及数组（但不是所有的对象实例都在堆中）**。其大小通过-Xms(最小值)和-Xmx(最大值)参数设置（**最大最小值都要小于1G**），前者为启动时申请的最小内存，默认为操作系统物理内存的1/64，后者为JVM可申请的最大内存,默认为物理内存的1/4，**默认当空余堆内存小于40%时**，JVM会**增大**堆内存到-Xmx指定的大小，可通过-XX:MinHeapFreeRation=来指定这个比列；当**空余堆内存大于70%**时，JVM会**减小**堆内存的大小到-Xms指定的大小，可通过XX:MaxHeapFreeRation=来指定这个比列，当然为了避免在运行时频繁调整Heap的大小，**通常-Xms与-Xmx的值设成一样**。堆内存 = 新生代+老生代+持久代。在我们垃圾回收的时候，我们往往将堆内存分成**新生代和老生代（大小比例1：2）**，新生代中由Eden和Survivor0，Survivor1组成，**三者的比例是8：1：1**，新生代的回收机制采用**复制算法**，在Minor GC的时候，我们**都留一个存活区**用来**存放存活的对象**，真正进行的区域是Eden+其中一个存活区，当我们的对象时长超过一定年龄时（默认**15**，可以通过参数设置），将会把对象放入老生代，当然大的对象会直接进入老生代。老生代采用的回收算法是**标记整理算法。**

###### 2. 方法区（Method Area）

方法区也称”**永久代**“，它用于**存储虚拟机加载的类信息、常量、静态变量**、是各个**线程共享的内存区域**。默认最小值为16MB，最大值为64MB（64位JVM由于指针膨胀，默认是85M），可以通过-XX:PermSize 和 -XX:MaxPermSize 参数限制方法区的大小。它是**一片连续的堆空间**，永久代的**垃圾收集是和老年代(old generation)捆绑在一起的**，因此**无论谁满**了，**都会触发永久代和老年代的垃圾收集**。不过，一个明显的问题是，当JVM加载的类信息容量超过了参数-XX：MaxPermSize设定的值时，应用将会报OOM的错误。参数是通过-XX:PermSize和-XX：MaxPermSize来设定的。

###### 3.虚拟机栈(JVM Stack)

描述的是**java方法执行的内存模型**：每个**方法被执行的时候都会创建一个”栈帧”**,用于**存储局部变量表(包括参数)、操作栈、方法出口**等信息。每个方法被调用到执行完的过程，就**对应着一个栈帧在虚拟机栈中从入栈到出栈的过程**。**声明周期与线程相同，是线程私有的**。栈帧由三部分组成：局部变量区、操作数栈、帧数据区。局部变量区被组织为以一个字长为单位、从0开始计数的数组，和局部变量区一样，操作数栈也被组织成一个以字长为单位的数组。但和前者不同的是，它不是通过索引来访问的，而是通过入栈和出栈来访问的，可以看作为临时数据的存储区域。除了局部变量区和操作数栈外，java栈帧还需要一些数据来支持常量池解析、正常方法返回以及异常派发机制。这些数据都保存在java栈帧的帧数据区中。

**局部变量表: 存放**了编译器可知的各种**基本数据类型**、对象引用(引用指针，并非对象本身)，其中64位长度的long和double类型的数据会占用2个局部变量的空间，其余数据类型只占1个。局部变量表**所需的内存空间**在**编译期间完成分配**，当进入一个方法时，这个方法需要在栈帧中**分配多大的局部变量是完全确定**的，在**运行期间**栈帧**不会改变局部变量表的大小空间**。

###### 4.本地方法栈(Native Stack)

与虚拟机栈基本类似，区别在于**虚拟机栈为虚拟机执行的java方法服务**，而**本地方法栈则是为Native方法服务**。(栈的空间大小远远小于堆)

###### 5.程序计数器（PC Register）

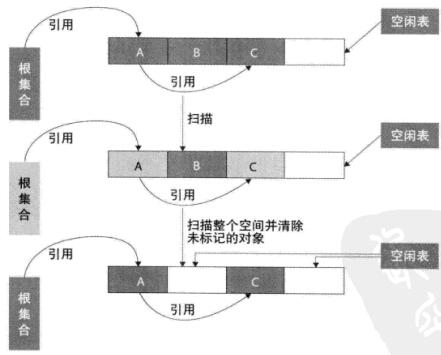
是**最小的一块内存区域**，它的**作用**是**当前线程所执行的字节码的行号指示器**，在虚拟机的模型里，字节码解释器工作时就是通过**改变这个计数器的值来选取下一条需要执行的字节码指令**，**分支**、**循环、异常处理、线程恢复**等基础功能都需要**依赖计数器**完成。

###### 6.直接内存

直接内存并**不是虚拟机内存的一部分**，也不是Java虚拟机规范中定义的内存区域。jdk1.4中新加入的NIO，引入了通道与缓冲区的IO方式，它可以调用Native方法直接分配堆外内存，这个堆外内存就是本机内存，不会影响到堆内存的大小.

### JVM垃圾回收算法

##### 1.标记清除



**原理**：

* 从根集合节点进行扫描，标记出所有的存活对象，最后扫描整个内存空间并清除没有标记的对象（即死亡对象）

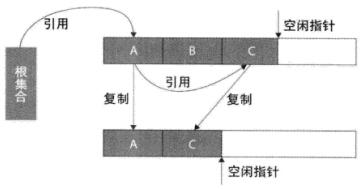
**适用场合**：

* 存活对象较多的情况下比较高效
* 适用于年老代（即旧生代）

**缺点：**

* 标记清除算法带来的一个问题是会存在大量的空间碎片，因为回收后的空间是不连续的，这样给大对象分配内存的时候可能会提前触发full gc。

##### 2.复制算法



**原理：**

* 从根集合节点进行扫描，标记出所有的存活对象，并将这些存活的对象复制到一块儿新的内存（图中下边的那一块儿内存）上去，之后将原来的那一块儿内存（图中上边的那一块儿内存）全部回收掉

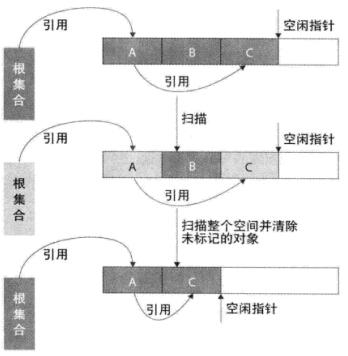
**适用场合：**

* 存活对象较少的情况下比较高效
* 扫描了整个空间一次（标记存活对象并复制移动）
* 适用于年轻代（即新生代）：基本上98%的对象是”朝生夕死”的，存活下来的会很少

**缺点：**

* 需要一块儿空的内存空间
* 需要复制移动对象

##### 3.标记整理



**原理：**

* 从根集合节点进行扫描，标记出所有的存活对象，最后扫描整个内存空间并清除没有标记的对象（即死亡对象）（可以发现前边这些就是标记-清除算法的原理），清除完之后，将所有的存活对象左移到一起。

**适用场合：**

* 用于年老代（即旧生代）

**缺点：**

* 需要移动对象，若对象非常多而且标记回收后的内存非常不完整，可能移动这个动作也会耗费一定时间
* 扫描了整个空间两次（第一次：标记存活对象；第二次：清除没有标记的对象）

优点：

* 不会产生内存碎片

##### 4.增量算法

增量算法的基本思想是，如果一次性将所有的垃圾进行处理，需要造成系统长时间的停顿，那么就可以让垃圾收集线程和应用程序线程交替执行。每次，垃圾收集线程只收集一小片区域的内存空间，接着切换到应用程序线程。依次反复，直到垃圾收集完成。使用这种方式，由于在垃圾回收过程中，间断性地还执行了应用程序代码，所以能减少系统的停顿时间。但是，因为线程切换和上下文转换的消耗，会使得垃圾回收的总体成本上升，造成系统吞吐量的下降。

### 垃圾回收器

##### 1.Serial收集器

Serial收集器是最古老的收集器，它的缺点是当Serial收集器想进行垃圾回收的时候，必须暂停用户的所有进程，即stop the world。到现在为止，它依然是虚拟机运行在client模式下的默认新生代收集器，与其他收集器相比，对于限定在单个CPU的运行环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾回收自然可以获得最高的单线程收集效率。

##### 2.ParNew收集器

ParNew收集器是Serial收集器新生代的多线程实现，注意在进行垃圾回收的时候依然会stop the world，只是相比较Serial收集器而言它会运行多条进程进行垃圾回收。

ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个CPU的环境中都不能百分之百的保证能超越Serial收集器。当然，随着可以使用的CPU的数量增加，它对于GC时系统资源的利用还是很有好处的。它默认开启的收集线程数与CPU的数量相同，在CPU非常多（譬如32个，现在CPU动辄4核加超线程，服务器超过32个逻辑CPU的情况越来越多了）的环境下，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

##### 3.Parallel Scavenge收集器

Parallel是采用复制算法的多线程新生代垃圾回收器，似乎和ParNew收集器有很多的相似的地方。但是Parallel Scanvenge收集器的一个特点是它所关注的目标是吞吐量(Throughput)。所谓吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间 / (运行用户代码时间 + 垃圾收集时间)。停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能够提升用户的体验；而高吞吐量则可以最高效率地利用CPU时间，尽快地完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

##### 4.CMS收集器

CMS(Concurrent Mark Swep)收集器是一个比较重要的回收器，现在应用非常广泛，我们重点来看一下，CMS一种获取最短回收停顿时间为目标的收集器，这使得它很适合用于和用户交互的业务。从名字(Mark Swep)就可以看出，CMS收集器是基于标记清除算法实现的。它的收集过程分为四个步骤：

1. 初始标记(initial mark)
2. 并发标记(concurrent mark)
3. 重新标记(remark)
4. 并发清除(concurrent sweep)

注意初始标记和重新标记还是会stop the world，但是在耗费时间更长的并发标记和并发清除两个阶段都可以和用户进程同时工作。

不过由于CMS收集器是基于标记清除算法实现的，会导致有大量的空间碎片产生，在为大对象分配内存的时候，往往会出现老年代还有很大的空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前开启一次Full GC。

为了解决这个问题，CMS收集器默认提供了一个-XX:+UseCMSCompactAtFullCollection收集开关参数（默认就是开启的)，用于在CMS收集器进行FullGC完开启内存碎片的合并整理过程，内存整理的过程是无法并发的，这样内存碎片问题倒是没有了，不过停顿时间不得不变长。虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction参数用于设置执行多少次不压缩的FULL GC后跟着来一次带压缩的（默认值为0，表示每次进入Full GC时都进行碎片整理）。

不幸的是，它作为老年代的收集器，却无法与jdk1.4中已经存在的新生代收集器Parallel Scavenge配合工作，所以在jdk1.5中使用cms来收集老年代的时候，新生代只能选择ParNew或Serial收集器中的一个。ParNew收集器是使用-XX:+UseConcMarkSweepGC选项启用CMS收集器之后的默认新生代收集器，也可以使用-XX:+UseParNewGC选项来强制指定它。

##### 5.G1收集器

G1收集器是一款面向服务端应用的垃圾收集器。HotSpot团队赋予它的使命是在未来替换掉JDK1.5中发布的CMS收集器。与其他GC收集器相比，G1具备如下特点：

1. 并行与并发：G1能更充分的利用CPU，多核环境下的硬件优势来缩短stop the world的停顿时间。
2. 分代收集：和其他收集器一样，分代的概念在G1中依然存在，不过G1不需要其他的垃圾回收器的配合就可以独自管理整个GC堆。
3. 空间整合：G1收集器有利于程序长时间运行，分配大对象时不会无法得到连续的空间而提前触发一次GC。
4. 可预测的非停顿：这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。

在使用G1收集器时，Java堆的内存布局和其他收集器有很大的差别，它将这个Java堆分为多个大小相等的独立区域，虽然还保留新生代和老年代的概念，但是新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。

虽然G1看起来有很多优点，实际上CMS还是主流。

### 与GC相关的常用参数

除了上面提及的一些参数，下面补充一些和GC相关的常用参数：

* -Xmx: 设置堆内存的最大值。
* -Xms: 设置堆内存的初始值。
* -Xmn: 设置新生代的大小。
* -Xss: 设置栈的大小。
* -PretenureSizeThreshold: 直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配。
* -MaxTenuringThrehold: 晋升到老年代的对象年龄。每个对象在坚持过一次Minor GC之后，年龄就会加1，当超过这个参数值时就进入老年代。
* -UseAdaptiveSizePolicy: 在这种模式下，新生代的大小、eden 和 survivor 的比例、晋升老年代的对象年龄等参数会被自动调整，以达到在堆大小、吞吐量和停顿时间之间的平衡点。在手工调优比较困难的场合，可以直接使用这种自适应的方式，仅指定虚拟机的最大堆、目标的吞吐量 (GCTimeRatio) 和停顿时间 (MaxGCPauseMills)，让虚拟机自己完成调优工作。
* -SurvivorRattio: 新生代Eden区域与Survivor区域的容量比值，默认为8，代表Eden: Suvivor= 8: 1。
* -XX:ParallelGCThreads：设置用于垃圾回收的线程数。通常情况下可以和 CPU 数量相等。但在 CPU 数量比较多的情况下，设置相对较小的数值也是合理的。
* -XX:MaxGCPauseMills：设置最大垃圾收集停顿时间。它的值是一个大于 0 的整数。收集器在工作时，会调整 Java 堆大小或者其他一些参数，尽可能地把停顿时间控制在 MaxGCPauseMills 以内。
* -XX:GCTimeRatio:设置吞吐量大小，它的值是一个 0-100 之间的整数。假设 GCTimeRatio 的值为 n，那么系统将花费不超过 1/(1+n) 的时间用于垃圾收集。

# 构架师晋级

## [架构师进阶之独孤九剑：设计模式详解](http://youzhixueyuan.com/detailed-design-patterns.html)



我们继续架构师进阶之独孤九剑进阶，目前我们仍然在第一阶段：**程序设计和开发环节**。

“

设计模式不仅仅只是一种规范，更多的是一种设计思路和经验总结，目的只有一个：提高你高质量编码的能力。以下主要分为三个环节：为什么要用设计模式、设计模式的分类，以及最后推荐大家三本设计模式相关的书籍，希望对你有所帮助。

### 为什么要使用设计模式

1、设计模式是前人根据经验总结出来的，使用设计模式，就相当于是站在了前人的肩膀上。

2、设计模式使程序易读。熟悉设计模式的人应该能够很容易读懂运用设计模式编写的程序。

3、设计模式能使编写的程序具有良好的可扩展性，满足系统设计的开闭原则。比如策略模式，就是将不同的算法封装在子类中，在需要添加新的算法时，只需添加新的子类，实现规定的接口，即可在不改变现有系统源码的情况下加入新的系统行为。

4、设计模式能降低系统中类与类之间的耦合度。比如工厂模式，使依赖类只需知道被依赖类所实现的接口或继承的抽象类，使依赖类与被依赖类之间的耦合度降低。

5、设计模式能提高代码的重用度。比如适配器模式，就能将系统中已经存在的符合新需求的功能代码兼容新的需求提出的接口 。

6、设计模式能为常见的一些问题提供现成的解决方案。

7、设计模式增加了重用代码的方式。比如装饰器模式，在不使用继承的前提下重用系统中已存在的代码。

### 设计模式分类和作用

**总体来说设计模式分为三大类：**



**创建型模式的作用：**

创建型模式，顾名思义就是创建对象的模式，对对象实例化过程进行了抽象。它们帮助一个系统独立于如何创建、组合和表示它的那些对象。

**创建型模式，共五种：**

* 工厂方法模式
* 抽象工厂模式
* 单例模式
* 建造者模式
* 原型模式

**1.工厂方法模式**：定义一个用于创建对象的接口，让子类决定实例化哪一个类，也就是将创建工作推迟到子类来完成。

**2.抽象工厂模式**：提供一个创建一系列相关或者相互依赖对象的接口，而无需指定它们（指的是这些相关或者相互依赖的对象）的具体的类。

**3.生成器模式**：将一个复杂对象的构建和它的表示相分离，使得同样的构建过程可以创建出不同的表示。

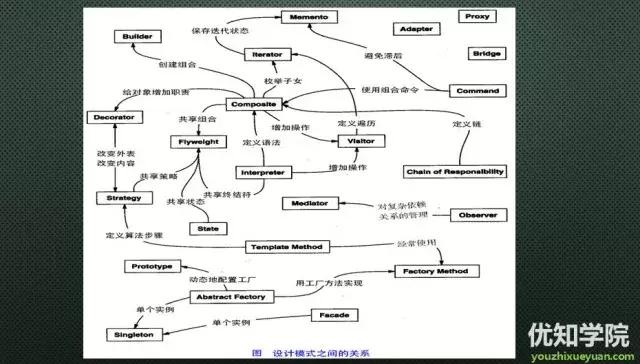
**4.原型模式**：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

**5.单例模式**：保证一个类仅有一个实例，并提供一个访问它的全局访问点。换句话来说就是，单例模式通过将单例类的构造器访问修饰符设置为private，禁止客户直接通过new的方式来实例化单例类；与此同时，单例类提供一个静态访问点来获取到通过内部创建出的唯一单例类对象。

**一般创建型设计模式面试必问：**

建议以下四类都熟练掌握

* 工厂方法模式
* 抽象工厂模式
* 单例模式
* 建造者模式



**结构型模式，共七种：**

* 适配器模式
* 装饰器模式
* 代理模式
* 外观模式
* 桥接模式
* 组合模式
* 享元模式

**结构型模式的作用：**

结构型模式主要用于描述如何组合类和对象以获得更大的结构。其中，结构型类模式采用继承机制来组合接口和实现，而结构型对象模式则采用组合/聚合方式来组合对象以实现新功能，因为它可以在运行时刻改变对象组合关系，所以对象组合方式具有更大的灵活性，这种机制是无法通过静态类组合来实现的。

**结构型模式简介：**

1.适配器模式：将一个类的接口转换成客户希望的另外一种接口，这样就能实现已有接口的复用。适配器主要有类适配器和对象适配器两种实现方式，通常情况下，推荐优先使用对象适配器方式。

2.桥接模式：将抽象部分与实现部分分离，使它们都可以独立地变化。它主要用于应对多维度变化点问题，通过对象组合的方式，可以极大地减少子类的数目，同时还能让不同维度独立扩展变化。

3.组合模式：将对象组合成树形结构以表示“整合-部分”的层次结构，从而使得用户对单个对象和组合对象的使用具有一致性，也就是客户端能够透明地无区别地操作两者。

4.装饰模式：动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式相比生成子类更为灵活。

5.外观模式：为子系统中的一组接口提供一个一致的接口，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。这样原来需要客户直接与复杂的子系统打交道、交互，现在这一过程将完全将交由外观对象来完成，极大地方便了客户端的调用。

6.享元模式：运用共享技术有效地支持大量细粒度的对象。享元模式关键是将对象的内部状态和外部状态分离，尽可能地对“稳定”的内部状态进行共享，而将会随运用场景而改变的状态通过外部状态传入。

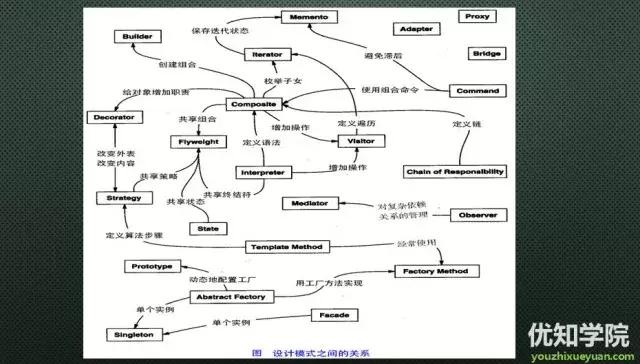
7.代理模式：为其他对象提供一种代理以控制对这个对象的访问。主要是在客户端和目标对象间增加一层间接层，通过这个间接层来完成对目标对象的种种控制操作，所以也就形成了不同功能类型的代理呢，比如远程代理、保护代理和虚代理等等。

**行为型模式，共十一种：**

* 策略模式
* 模板方法模式
* 观察者模式
* 迭代子模式
* 责任链模式
* 命令模式
* 备忘录模式
* 状态模式
* 访问者模式
* 中介者模式
* 解释器模式

**行为模式的作用：**

行为型模式设计到算法和对象间的职责分配，不仅描述对象或类的模式，还描述它们之间的通信方式，刻划了运行时难以跟踪的复杂的控制流，它们将你的注意力从控制流转移到对象间的关系上来。



### 设计模式书籍推荐

**第一本：Head First设计模适合式**

适合：设计模式入门



**第二本：设计模式**

适合：有一定的设计模式基础之后再看



**第三本：敏捷软件开发**

适合：对设计模式以及敏捷的了解，名字叫敏捷软件开发，实则主要讲解了大量设计模式。



推荐以上的三本设计模式相关的书籍，顺序可以从上往下看，出入门到深入，也许效果会更好。

## [**UML是什么?UML常用图以及建模工具有哪些?**](http://youzhixueyuan.com/what-is-umluml-used-diagrams-and-uml-modeling-tools.html)



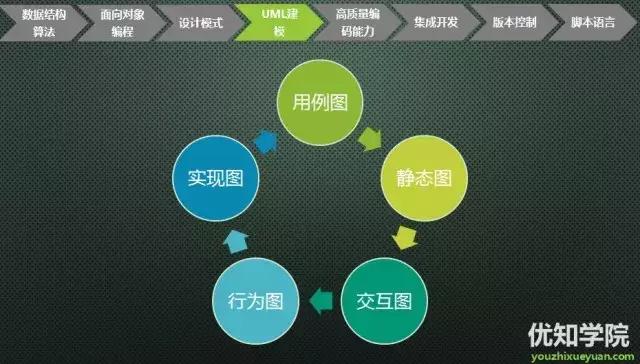
“

在做项目设计方案的时候，理解为需求后，我们都会做技术设计方案，这个时候就需要用到UML建模，涉及到UML常用图形以及工具画图，以下我会详细介绍UML图形和我个人推荐的UML建模工具。

UML即Unified Model Language，是一种建模语言，也是标准建模语言。在软件开发中，当系统规模比较复杂时，需要用图形抽象地来表达复杂的概念，让整个软件设计更具有可读性，可理解性，以便尽早发现软件设计时存在的潜在问题，从而降低开发风险。同时，也极大地方便了业务人员与开发人员之间的交流。

### UML常用建模图

**UML标准建模语言中5种主要的类型图：**



1、用例图：指的是从用户角度来描述系统功能，并且指明各功能操作者。

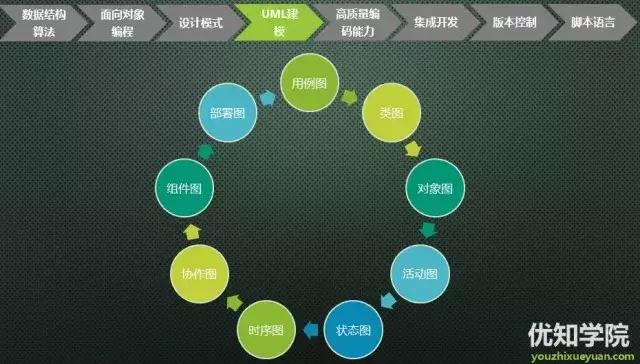
2、静态图：包含的是类图和对象图。类图是一种静态模型类型，是用来表示类之间的联系、类的属性以及操作，在系统的整个生命周期都是有效的。对象图则是类图的一个实例，使用的标识几乎与类图一致，但是其生命周期有限，只能在系统中某一时间段内存在。

3、交互图：包含的是时序图和协作图，是用来描述对象之间的交互关系。时序图强调的是对象之间的消息发送顺序，是对象之间动态合作关系。协作图则是用来描述对象之间的协作关系，既显示对象间的动态合作关系，又显示对象以及它们之间的关系。时序图用来强调时间和顺序，协作图则用来强强调上下级的关系。

4、行为图：包含的是活动图和状态图，是用来描述系统的动态模型于组成对象之间的交互关系。活动图描述的是为了满足用例要求所进行的活动以及活动间的约束关系，方便识别并进行活动。状态图是类的补充，是用来描述类的对象所有可能的状态以及事件发生时状态的转移条件。

5、实现图：包含的是部署图和组件图。部署图是用来表示建模系统的物理部署。组件图则用来表示建模软件的组织以及其相互之间的关系。

### UML建模时常见的9种图：



**1、用例图：**

用例图是需求分析的产物，主要是用来描述用户是如何使用一个系统的，是用户所能观察和使用到的系统功能的模型图。

**2、类图：**

UML类图，是用来描述系统中的类以及各个类之间的关系。系统中可以有多个类图，单个类图则只是表达了系统的一个方面。类图可以帮助我们在正确编写代码之前对系统有个很全面的认识，是建模中最常见的一种类型图。

**3、对象图：**

对象图描述的是一组对象之间的关系，而不是类之间的关系。它是类图的变体，但是又与之不同，对象显示的类的多个对象而不是实际的类。

**4、活动图：**

UML活动图，顾名思义就是UML的动态模型的一种图形，对于系统的功能建模特别重要。它用来描述满足用例要求所要进行的活动以及活动间的约束关系，有利于识别并行活动。简单来讲，就是表示在处理某个活动时，两个或者更多类对象之间的过程控制流。

**5、状态图：**

UML状态图，用于显示状态机，即描述一个对象所处的可能状态以及状态之间的转移。用状态图建模可以帮助开发人员分析复杂对象的各种状态的转换，以及对象何时执行怎样的动作。

**6、时序图：**

UML时序图，又叫序列图或者顺序图，是一种用来描述对象之间传送消息的时间顺序，是用来表示用例中的行为顺序。

**7、协作图：**

UML协作图(又叫通信图)是一种作用于显示对象之间如何进行交互以执行特点用例或用例中特点部分行为的交互图，它强调的是发送和接收消息的对象之间的组织结构。

**8、组件图：**

UML组件图(又叫构件图)，是用来描述在软件系统中遵从并实现一组接口的物力的、可替换的软件模块。它所表现的是一种系统静态实现的结构，能够帮助开发人员对系统组成达成一致的认识。

**9、部署图：**

UML 部署图是一个用来描述系统的硬件配置和部署以及软件的构件和模块在不同节点上分布的模型图。它能够帮助系统相关人员了解系统中各个构件部署在什么硬件上以及硬件之间的交互关系。

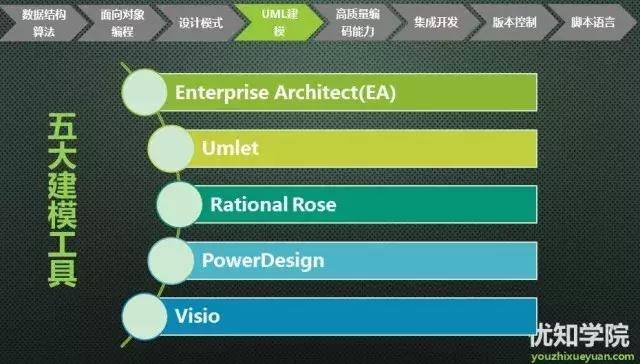
**UML中的各种图之间的区别：**

1、各有其侧重点：比如说用例图侧重于描述用户的需求，而类图则侧重于描述系统的具体实现。

2、所描述的方面各不相同：比如类图是用来描述系统结构的，序列图则是用来描述系统行为的。

3、抽象的层次也不尽相同：比如头见图是用来描述系统的模块结构，抽象层次比较高，而对象图则用来描述具体的模块实现，抽象层次就比较低。

### UML工具



**Visio**

Visio 原来仅仅是一种画图工具，能够用来描述各种图形（从电路图到房屋结构图），也是到VISIO2000才开始引进软件分析设计功能到代码生成的全部功能，它可以说是目前最能够用图形方式来表达各种商业图形用途的工具。

**Rational Rose**

ROSE是直接从UML发展而诞生的设计工具，它的出现就是为了对UML建模的支持，ROSE一开始没有对数据库端建模的支持，但是在现在的版本中已经加入数据库建模的功能。ROSE主要是在开发过程中的各种语义、模块、对象以及流程，状态等描述比较好，主要体现在能够从各个方面和角度来分析和设计，使软件的开发蓝图更清晰，内部结构更加明朗（但是它的结构仅仅对那些对掌握UML的开发人员，也就是说对客户了解系统的功能和流程等并不一定很有效），对系统的代码框架生成有很好的支持。但对数据库的开发管理和数据库端的迭代不是很好。

**PowerDesign**

PowerDesigner原来是对数据库建模而发展起来的一种数据库建模工具。直到7.0版才开始对面向对象的开发的支持，后来又引入了对UML的支持。但是由于PowerDesigner侧重不一样，所以它对数据库建模的支持很好，支持了能够看到的90%左右的数据库，对UML的建模使用到的各种图的支持比较滞后。但是在最近得到加强。

**Enterprise Architect（EA）**

Enterprise Architect是一个全功能的、基于UML的visual CASE工具，主要用于设计、编写、构建并管理以目标为导向的软件系统。它支持用户案例、商务流程模式以及动态的图表、分类、界面、协作、结构以及物理模型。此外，它还支持C++、Java、Visual Basic、Delphi、C#以及VB.Net。

**这个是我用过的最好的UML工具之一，个人推荐使用。**

**Umlet**

开源的免费UML工具，这个软件实际上是用JAVA写的，还有有一套简化的语言，提供用编码的方式画时序图等各种图，也可以用code的方式自定义页面元素。

**最后，UML建模除了画图本身以外，最重要的是理清整个设计方案思路，这一点才UML画图的精髓所在。**

## 从程序员进阶到架构师，6大核心技能要领详解



“

[java架构师](http://youzhixueyuan.com/tag/java%e6%9e%b6%e6%9e%84%e5%b8%88)技能将分为如下6大环节：数据结构和算法，Java高级特性，Java web核心，数据库，Java框架与必备工具，系统架构设计。

希望能真正帮助到从程序员进阶到架构师之路的朋友。

### 数据结构和算法



**算法分析与计算**

算法时间复杂度和空间复杂度的分析计算

**算法思想**

递推、递归、穷举、贪心、分治、动态规划、迭代、分枝界限

**常用数据结构**

数组、链表、堆、栈、队列、Hash表、二叉树等

**经典算法**

**排序**

经典排序：插入排序、冒泡排序、快排（分划交换排序）、直接选择排序、堆排序、合并排序等

**查找**

经典查找：顺序查找、二分查找、二叉排序树查找

### java语言高级特性



**深入理解面向对象**

面向对象的基本思想是使用类、对象、继承、封装、消息等基本概念来进行程序设计。面向对象的方式实际上由OOA（面向对象分析）、OOD（面向对象设计）和OOP（面向对象编程）三个部分有机组成。

对象的三大特性：封装、继承和多态，优缺点

如何设计类，类的设计原则以及构造函数，内部类，抽象类，接口，对象的多态性，接口和抽象类的区别。

**理解异常处理**

Java 常见异常种类

Java Exception

Error

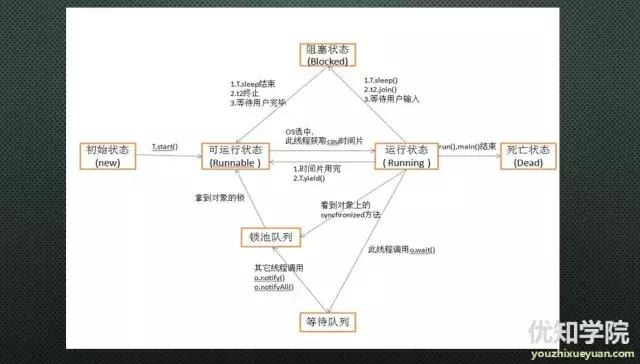
Runtime Exception 运行时异常

Exception

throw 用户自定义异常

java标准里也提到过，希望能用异常来处理错误信息以及后续流程，所以异常不仅仅只是异常，而是一个标准错误处理机制，并且也鼓励这样使用错误信息。

**多线程**



**java多线程5大状态：**

* 新建状态（New）
* 就绪状态（Runnable）
* 运行状态（Running）
* 阻塞状态（Blocked）
* 死亡状态（Dead）

**Java多线程掌握**

* 创建与启动
* 线程和进程的概念
* 线程安全
* 线程之间的通讯
* 线程的同步与锁
* 死锁问题的剖析
* 线程生命周期
* 线程池

**输入与输出**

标准输入输出，文件的操作，网络上的数据流，字符串流，对象流，zip文件流等等，java中将输入输出抽象称为流，就好像水管，将两个容器连接起来。

Java I/O主要包括如下几个层次，包含三个部分：

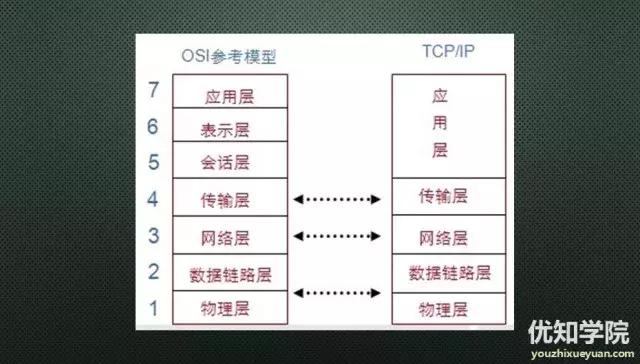
* 流式部分――IO的主体部分
* 非流式部分――主要包含一些辅助流式部分的类，如：File类、RandomAccessFile类和FileDescriptor等类
* 其他类–文件读取部分的与安全相关的类

**反射**

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

涉及到类加载机制原理、反射构造方法、字段、方法、Properties配置文件等相关学习要领。

**网络编程**



七层模型与协议的对应关系、Socket原理机制、UDP、TCP传输等知识点。

**JVM**

涉及到程序进阶，除了工作中平常使用以外，一定要深入去理解使用之外的核心原理，JVM特别是重中之重，JVM内存划分、class加载机制以及GC策略内存划分，Young Generation（年轻代）、Old G/eneration（年老代）以及Perm Generation（永久代）等。

### java web开发



**前段技术**

如今H5特别火爆，上图我截取了部分H5相关的重点掌握：html5、css3、js、以及对应的框架jquery、node等。

平时多掌握点web开发相关的，非常方便你今后后端调试等，前段知识还是要跟上。

**Java Web核心**

这里还是需要掌握整个java web从容器启动到request、filter、listener的过程，最好自己设置断点调试，把完整的生命周期都走一次。

**模板引擎**

上图是常见的模板引擎，比如淘宝一直使用velocity，现在最新又出来了个新模板引擎：beetl,有兴趣的同学也可以自己发掘。

**高级特性**

* SSI技术
* 安全JCCA/JAAS
* 通信JNDI/JMS/JavaMain/JAF
* 集成JCA
* 事务JTA

### 数据库



首先，数据库设计的基本原则还是需要掌握

**数据库设计原则和范式**

第一范式，确保每列保持原子性。

第二范式，确保表中的每列都和主键相关。

第二范式，在第一范式的基础之上更进一层。

第三范式，确保每列都和主键列直接相关,而不是间接相关。

**Sql与NoSql的优缺点以及使用场景分析**

SQL 关系型数据库：SQL Server，Oracle，MySQL(开源)，PostgreSQL(开源)

NoSQL泛指非关系型数据库 ：MongoDB，Redis，Memcached,Hbase,CouchDB。

**大型互联网项目常用的数据库选型方案：**

采用MySQL + NoSQL的组合方案，根据业务场景和数据访问量来分别采用。

**关系式数据库必备**

事务（ACID、工作原理、事务的隔离级别、锁、事务的传播机制）

数据库创建，权限分配，表的创建，增删改查，连接，子查询

索引、触发器、存储过程、事务控制等

**数据库性能优化**

索引原理及适用，大表查询优化，多表连接查询优化，子查询优化

分库、分表、备份、迁移、导入，冷备热备，主从备份、双机热备、纵向扩展、横向扩展等这些都是属于比较常见的数据库方案，**我在淘宝具体挑战性的一次架构演变中一文中**谈到了数据库纵向和横向的发展策略，有兴趣的同学可以翻看历史查看。

### java框架与工具



**web开发框架**

Struts2、SpringMVC、spring、hibernate、myBatis

在搭建日志：Log4j

单元测试：JUnit

**构建工具**

推荐nexus搭建一套自己的代码仓库中心，采用maven管理，这些都是比较常见而且有效的构建方案。

**Web服务器**

* Tomcat
* JBoss
* Jetty
* Resin
* WebLogic
* WebSphere等

**通信协议**

* RMI
* Hessia
* Burlap
* Httpinvoker
* Soap
* WebService等

**工作流**

Activity、JBPM

**搜索引擎**

lucene，基于lucene封装的solr

### 大型网站架构设计



其实就是要很清楚整个技术架构的演变历程，知道每个阶段的瓶颈在哪里，以及对应的解决方案，大型网站架构设计一般包含如下：

**搭建分布系统的基础设施**

**缓存搭建**

分布式缓存搭建 memcached ,redis（推荐），动态、静态数据的缓存，以及配合单点登录的使用等。

**负载均衡**

Nginx/HaProxy

**CDN搭建**

为了应付复杂的网络环境和不同地区用户的访问，通过CDN和反向代理加快用户访问的速度，同时减轻后端服务器的负载压力。CDN与反向代理的基本原理都是缓存。

**分布式储存搭建**

常见的分布式文件系统有，GFS、HDFS、Lustre 、Ceph 、GridFS 、mogileFS、TFS、FastDFS等，比如GFS（Google File System）,TFS(Taobao File System),tfs参考tfs而来，所以名字也直接参考了。

**消息系统搭建**

目前使用较多的消息队列有ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，RocketMQ，优先推荐ActiveMQ。

**消息系统使用四大场景**

* 异步处理
* 应用解耦
* 流量削锋
* 消息通讯

**常用的大数据方案**

一般先分库，如果分库后查询仍然慢，于是按照分库的思想开始做分表。

比如淘宝中期数据库压力非常大了，于是数据库端按照业务做垂直拆分：交易数据库、用户数据库、商品数据库、店铺数据库等进行拆分。

**采用sql和nosql混搭搭建再配合搜索引擎**

随着业务越来越复杂，对数据存储和检索的需求也越来越复杂，系统需要采用一些非关系型数据库如NoSQL和分数据库查询技术如搜索引擎。应用服务器通过统一数据访问模块访问各种数据，减轻应用程序管理诸多数据源的麻烦。

**除此之外，还要考虑安全以及机房容灾以及系统运维监控等。**

## java架构师进阶之独孤九剑：数据结构以及书籍推荐

java-programmers-data-structure-in-the-algorithm.html



这是整个java架构师连载系列，分为9大步骤，我们现在还在第一个步骤：程序设计和开发->数据结构与算法。

“

如果说 Java 是自动档轿车，C 就是手动档吉普。数据结构呢？是变速箱的工作原理，

你完全可以不知道变速箱怎样工作，就把自动档的车子从 A 开到 B，而且未必就比懂得的人慢。写程序这件事，和开车一样，经验可以起到很大作用，但如果你不知道底层是怎么工作的，就永远只能开车，既不会修车，也不能造车。如果你对这两件事都不感兴趣也就罢了，数据结构懂得用就好。但若你此生在编程领域还有点更高的追求，数据结构是绕不开的课题。

**不想学好基础的程序员很难进阶到架构师。**

### 为什么要学习数据结构

学习数据结构，并不仅仅是学习其中现成的那些队列，堆栈，二叉树，图等经典结构， 也不仅仅是学习其中的那些快速排序、冒泡排序等算法。

更重要的是你要学习一种思想：如何把现实问题转化为计算机语言的表示。



当你用着java里面的容器类很爽的时候，你有没有想过，怎么ArrayList就像一个无限扩充的数组，也好像链表之类的。好用吗？好用，这就是数据结构的用处，只不过你在不知不觉中使用了。

校招会发现大公司考的就是这类的题目，刚开始不会考你java的线程，容器，多态什么的特性，考的就是你的基础，你的这些基础扎实，学其他不是问题。

用于现实世界的存储，我们使用的工具和建模。每种数据结构有自己的优点和缺点，想想如果Google的数据用的是数组的存储，我们还能方便地查询到所需要的数据吗。

而算法，在这么多的数据中如何做到最快的插入，查找，删除，也是在追求更快。

### 常用的数据结构



**链表**

链表是一种由节点（Node）组成的线性数据集合，每个节点通过指针指向下一个节点。它是一种由节点组成，并能用于表示序列的数据结构。

单链表：每个节点仅指向下一个节点，最后一个节点指向空（null）。

双链表：每个节点有两个指针p，n。p指向前一个节点，n指向下一个节点；最后一个节点指向空。

循环链表：每个节点指向下一个节点，最后一个节点指向第一个节点。

时间复杂度：

索引：O(n)

查找：O(n)

插入：O(1)

删除：O(1)

**栈**

栈是一个元素集合，支持两个基本操作：push用于将元素压入栈，pop用于删除栈顶元素。

后进先出的数据结构（Last In First Out, LIFO）

时间复杂度

索引：O(n)

查找：O(n)

插入：O(1)

删除：O(1)

**队列**

队列是一个元素集合，支持两种基本操作：enqueue 用于添加一个元素到队列，dequeue 用于删除队列中的一个元素。

先进先出的数据结构（First In First Out, FIFO）。

时间复杂度

索引：O(n)

查找：O(n)

插入：O(1)

删除：O(1)

**树**

树是无向、联通的无环图。

**二叉树**

二叉树是一个树形数据结构，每个节点最多可以有两个子节点，称为左子节点和右子节点。

满二叉树（Full Tree）：二叉树中的每个节点有 0 或者 2 个子节点。

完美二叉树（Perfect Binary）：二叉树中的每个节点有两个子节点，并且所有的叶子节点的深度是一样的。

完全二叉树：二叉树中除最后一层外其他各层的节点数均达到最大值，最后一层的节点都连续集中在最左边。

二叉查找树

二叉查找树（BST）是一种二叉树。其任何节点的值都大于等于左子树中的值，小于等于右子树中的值。

时间复杂度

索引：O(log(n))

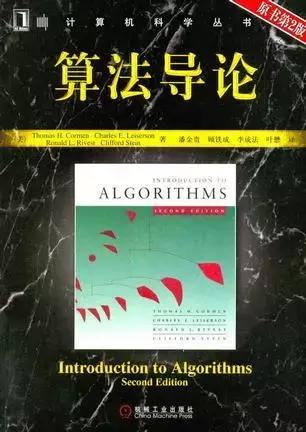
查找：O(log(n))

插入：O(log(n))

删除：O(log(n))

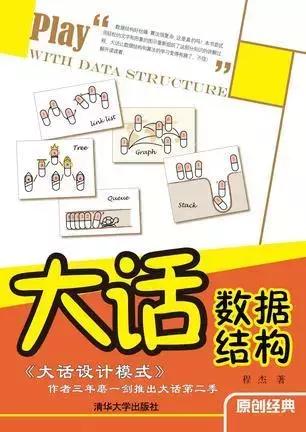
### 数据结构和算法书籍推荐

**1.算法导论**



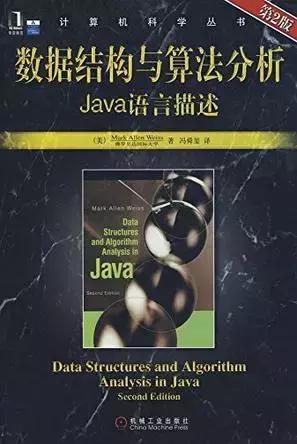
这本书深入浅出，全面地介绍了计算机算法。对每一个算法的分析既易于理解又十分有趣，并保持了数学严谨性。本书的设计目标全面，适用于多种用途。涵盖的内容有：算法在计算中的作用，概率分析和随机算法的介绍。书中专门讨论了线性规划，介绍了动态规划的两个应用，随机化和线性规划技术的近似算法等，还有有关递归求解、快速排序中用到的划分方法与期望线性时间顺序统计算法，以及对贪心算法元素的讨论。此书还介绍了对强连通子图算法正确性的证明，对哈密顿回路和子集求和问题的NP完全性的证明等内容。全书提供了900多个练习题和思考题以及叙述较为详细的实例研究。

**2.大话数据结构**



通篇以一种趣味方式来叙述，大量引用了各种各样的生活知识来类比，并充分运用图形语言来体现抽象内容，对数据结构所涉及到的一些经典算法做到逐行分析、多算法比较。与市场上的同类数据结构图书相比，本书内容趣味易读，算法讲解细致深刻，是一本非常适合自学的读物。

**3.数据结构与算法**



本书是国外数据结构与算法分析方面的经典教材，使用卓越的Java编程语言作为实现工具讨论了数据结构（组织大量数据的方法）和算法分析（对算法运行时间的估计）。 随着计算机速度的不断增加和功能的日益强大，人们对有效编程和算法分析的要求也不断增长。本书把算法分析与最有效率的Java程序的开发有机地结合起来，深入分析每种算法，内容全面、缜密严格，并细致讲解精心构造程序的方法。

## java架构师进阶之独孤九剑(一)-算法思想与经典算法

java-programmers-advanced-algorithm-ideas.html



“

这是整个架构师连载系列，分为9大步骤，我们现在还在第一个步骤：程序设计和开发->数据结构与算法。

我们今天讲解重点讲解算法。

### [算法思想](http://youzhixueyuan.com/tag/%e7%ae%97%e6%b3%95%e6%80%9d%e6%83%b3)



**1 贪心思想**

顾名思义，贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。当然，希望贪心算法得到的最终结果也是整体最优的。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。如单源最短路经问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。在面临选择时，贪心算法都作出对眼前来讲最有利的选择，不考虑对将来的不良影响，每个选择一旦做出，不可更改，不允许回溯，根据不同的贪心策略，贪心算法就不同，贪心解的质量也不同，所以贪心策略很重要。可以看出，此[算法思想](http://youzhixueyuan.com/tag/%e7%ae%97%e6%b3%95%e6%80%9d%e6%83%b3)很简单，具有高效性，但不一定得出最优解。

**2 分治法**

当我们求解某些问题时，由于这些问题要处理的数据相当多，或求解过程相当复杂，使得直接求解法在时间上相当长，或者根本无法直接求出。对于这类问题，我们往往先把它分解成几个子问题，找到求出这几个子问题的解法后，再找到合适的方法，把它们组合成求整个问题的解法。如果这些子问题还较大，难以解决，可以再把它们分成几个更小的子问题，以此类推，直至可以直接求出解为止。这就是分治策略的基本思想。

**3 动态规划**

动态规划算法通常用于求解具有某种最优性质的问题。在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值的解。动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。我们可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。

**4 搜索法**

搜索法包含穷举搜索，深度优先搜索，广度优先搜索，回溯法，分支限界法，其实这些算法的基础就是穷举搜索，只是加上一定的原则来优化过程，就形成了后面的几种算法，回溯法就是在深度优先搜索的基础上允许回溯，分支限界法是在广度优先搜索基础上允许剪枝，学习时主要学习思想，这些算法名字不要太在意。

**5 新近出现的部分算法简介**

**①　遗传算法**

从达尔文的生物进化论中得到启发，借鉴自然选择和进化的原理，模拟生物在自然界的进化过程所形成的一种优化求解方法，遗传算法从代表问题的可能潜在解集的一个种群出发，一个种群有一定数量的个体组成，每个个体实际上是染色体带有特征的实体，每一代根据个体的适应度大小挑选个体，并借助遗传算子进行交叉和变异，得到近似最优解。

**②　模拟退火算法**

他的出发点是物理中固体的退火过程与一般组合优化之间的相似性，固态物质退火时，通常先加温，使其中的粒子自由游动，然后逐渐降低温度，粒子也逐渐形成低能态的晶格，最终形成最低能量的基态。所以他从某一较高初温开始，伴随温度参数的不断下降重复抽样，最终得到全局最优解，他是基于概率的。

**③　蚁群算法**

蚁群算法是模拟自然界蚂蚁觅食过程的一种分布式，启发式群体智能算法，用于求解复杂的组合优化问题，如TSP,JSSP,GCP等问题。

**6 递归法**

所谓递归，就是指如果需要求解当前状态就需要求解其依赖的迁移状态。

一般来说，递归需要有边界条件、递归前进段和递归返回段。当边界条件不满足时，递归前进；当边界条件满足时，递归返回。

采用递归描述的算法通常有这样的特征：

1）为求解规模为N的问题，设法将它分解成规模较小的问题；

2）然后从这些小问题的解方便地构造出大问题的解，并且这些规模较小的问题也能采用同样的分解和综合方法，分解成规模更小的问题，并从这些更小问题的解构造出规模较大问题的解。

3）这样的分解方法具有收敛性。即存在一个递归返回状态。

**7 迭代法**

也称辗转法，是一种不断用变量的旧值递推新值的过程。

最常见的迭代法是牛顿法。其他还包括最速下降法、共轭迭代法、变尺度迭代法、最小二乘法、线性规划、非线性规划、单纯型法、惩罚函数法、斜率投影法、遗传算法、模拟退火等等。

利用迭代算法解决问题，需要做好以下三个方面的工作：

1）迭代变量：在可以用迭代算法解决的问题中，至少存在一个直接或间接地不断由旧值递推出新值的变量，这个变量就是迭代变量。

2）迭代关系：指如何从变量的前一个值推出其下一个值的公式（或关系）。迭代关系式的建立是解决迭代问题的关键，通常可以使用递推或倒推的方法来完成。

3）迭代过程：迭代过程的控制通常可分为两种情况：一种是所需的迭代次数是个确定的值，可以计算出来；另一种是所需的迭代次数无法确定。对于前一种情况，可以构建一个固定次数的循环来实现对迭代过程的控制；对于后一种情况，需要进一步分析出用来结束迭代过程的条件。

迭代法都可以转为递归法。迭代法可以理解为具有递归性质的非递归解法。当然，非递归解法还可利用栈的思想。

### 经典算法



**排序算法**

排序大的分类可以分为两种：内排序和外排序。在排序过程中，全部记录存放在内存，则称为内排序，如果排序过程中需要使用外存，则称为外排序。下面讲的排序都是属于内排序。

**内排序有可以分为以下几类：**

(1)、插入排序：直接插入排序、二分法插入排序、希尔排序。

(2)、选择排序：简单选择排序、堆排序。

(3)、交换排序：冒泡排序、快速排序。

(4)、归并排序

(5)、基数排序



**查找算法**

1. 顺序查找

2. 二分查找

3. 插值查找

4. 斐波那契查找

5. 树表查找

6. 分块查找

7. 哈希查找

查找是在大量的信息中寻找一个特定的信息元素，在计算机应用中，查找是常用的基本运算，例如编译程序中符号表的查找。以上是常见的七种查找算法，说是七种，其实二分查找、插值查找以及斐波那契查找都可以归为一类——插值查找。插值查找和斐波那契查找是在二分查找的基础上的优化查找算法。

## [**java架构师进阶独孤九剑系列（一）：数据结构与算法**](http://youzhixueyuan.com/java-programmers-advanced-data-structures-and-algorithms.html)

java-programmers-advanced-data-structures-and-algorithms.html



“

架构师进阶系列之独孤九剑，这套秘籍如果你能完整学会，年薪至少60万起，今天是架构师进阶连载开启篇。

### 完整的架构师进阶步骤

未来我会用1-2个月在实战为大家讲解架构师的九大必备技能。



### 架构师进阶系列之独孤九剑九大步骤：

**1-程序设计和开发**

2-操作系统要求

3-Java开发

4-Web开发

5-网络安全

6-数据库设计

7-架构设计要求

8-运维能力要求

9-产品业务要求

**1-程序设计和开发**

•1-1 [数据结构](http://youzhixueyuan.com/tag/%e6%95%b0%e6%8d%ae%e7%bb%93%e6%9e%84)[算法](http://youzhixueyuan.com/tag/%e7%ae%97%e6%b3%95)

•1-2 面向对象编程

•1-3 设计模式

•1-4 UML建模

•1-5 高质量编码能力

•1-6 集成开发

•1-7 版本控制

•1-8 脚本语言

**1-1**[**数据结构**](http://youzhixueyuan.com/tag/%e6%95%b0%e6%8d%ae%e7%bb%93%e6%9e%84)**和**[**算法**](http://youzhixueyuan.com/tag/%e7%ae%97%e6%b3%95)



**1-1-1**[**算法**](http://youzhixueyuan.com/tag/%e7%ae%97%e6%b3%95)**分析**

时间复杂度和空间复杂度

**1-1-2**[**算法**](http://youzhixueyuan.com/tag/%e7%ae%97%e6%b3%95)**思想**

递推、递归、穷举、贪心、分治、动态规划、迭代、分枝界限

**1-1-3**[**数据结构**](http://youzhixueyuan.com/tag/%e6%95%b0%e6%8d%ae%e7%bb%93%e6%9e%84)

数组、链表、堆、栈、队列、Hash表、二叉树等

**1-1-4 经典**[**算法**](http://youzhixueyuan.com/tag/%e7%ae%97%e6%b3%95)

**排序**

经典排序：插入排序、冒泡排序、快排（分划交换排序）、直接选择排序、堆排序、合并排序等

**查找**

经典查找：顺序查找、二分查找、二叉排序树查找

**1-1-5高级**[**算法**](http://youzhixueyuan.com/tag/%e7%ae%97%e6%b3%95)

贪婪

回溯

剪枝

动态规划

**1-1-6大数据算法**

hash分桶

统计

明天将正式开启架构师实战进阶系列，从 **1-1-1 算法分析与技术开始实战讲解。**

## 如何学习获得高并发的经验？

/how-to-get-high-concurrency-experience.html

### 什么是高并发

高并发是指在同一个时间点，有很多用户同时访问URL地址，比如：淘宝的双11、双12，就会产生高并发。又如贴吧的爆吧，就是恶意的高并发请求，也就是DDOS攻击。

**专业的角度什么是高并发**

高并发（High Concurrency）是互联网分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计保证系统能够同时并行处理很多请求。

高并发相关常用的一些名词，**高并发用户数，响应时间（Response Time），吞吐量（Throughput），TPS(Transactions Per Second)每秒事务数,QPS（Query Per Second）每秒查询率**等。

**响应时间**：系统对请求做出响应的时间。例如系统处理一个HTTP请求需要200ms，这个200ms就是系统的响应时间。

**并发用户数**：指的是现实系统中操作业务的用户，例如在淘宝双11疯狂抢单，同时在线量一定程度上代表了系统的并发用户数。

在性能测试工具中，一般称为**虚拟用户数(Virutal User)**，注意并发用户数跟注册用户数、在线用户数有很大差别的，并发用户数一定会对服务器产生压力的，而在线用户数只是 ”挂” 在系统上，对服务器 不产生压力，注册用户数一般指的是数据库中存在的用户数。

**吞吐量**：单位时间内处理的请求数量。

**TPS**：是Transactions Per Second的缩写，也就是事务数/秒。它是软件测试结果的测量单位。一个事务是指一个客户机向服务器发送请求然后服务器做出反应的过程。客户机在发送请求时开始计时，收到服务器响应后结束计时，以此来计算使用的时间和完成的事务个数。

**QPS**：Queries Per Second意思是“每秒查询率”，是一台服务器每秒能够相应的查询次数，是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。

Qps基本类似于Tps，但是不同的是，对于一个页面的一次访问，形成一个Tps；但一次页面请求，可能产生多次对服务器的请求，服务器对这些请求，就可计入“Qps”之中。

**一般的，评价系统性能均以每秒钟完成的技术交易的数量来衡量。系统整体处理能力取决于处理能力最低模块的TPS值。**

在做性能压力测试的时候，最重要的一个数据就是**TPS，**TPS可以通过PV进行换算，PV->TPS转换模型，具体换算发请参考**淘宝性能测试白皮书**里面就有非常详细的描述。

### 高并发解决方案

**1、服务器架构**

业务从发展的初期到逐渐成熟，服务器架构也是从相对单一到集群，再到分布式服务。

一个可以支持高并发的服务少不了好的服务器架构，需要有均衡负载，数据库需要主从集群，NoSQL缓存需要主从集群，静态文件需要上传CDN，这些都是能让业务程序流畅运行的强大后盾。

服务器这块多是需要运维人员来配合搭建，具体我就不多说了，点到为止。

大致需要用到的服务器架构如下：

**服务器**

均衡负载(如：nginx，阿里云SLB)

资源监控

分布式

**数据库**

主从分离，集群

DBA 表优化，索引优化等

分布式

**NoSQ**L

Redis

主从分离，集群

MongoDB

主从分离，集群

memcache

主从分离，集群

**消息队列**

秒杀、秒抢等活动业务，用户在瞬间涌入产生高并发请求，主要解决异步消息

**CDN**

以下静态资源需要转移到CDN基站

html

css

js

image

等等..

### 高并发学习经验

要想获得高并发学习经验，无非就两条：学习和实践。

学习，首先要正确了解高并发的知识点，比如上面我提到的这些内容。

其次，如果没有机会实践，完全可以自己模拟这样的业务场景，利用**并发测试工具。**

测试高并发可以使用第三方服务器或者自己测试服务器，利用测试工具进行并发请求测试，分析测试数据得到可以支撑并发数量的评估，这个可以作为一个预警参考，俗话说知己自彼百战不殆。

**第三方服务:**

阿里云性能测试

并发测试工具：

Apache JMeter

LoadRunner性能负载测试等。

高并发的内容今天就写到着，高并发相关的**架构演变过程**，还可以参考我的另外一篇文章：**最具决定性的淘宝架构演变**，里面具体涉及到的网站架构演变过程，看完后你对高并发会有更好的认识。

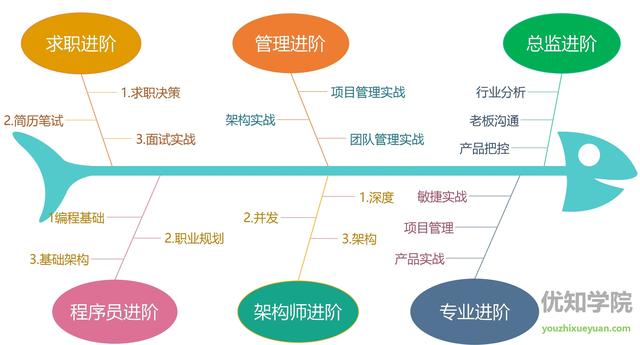
### 构架师晋级必读书单



以下是我推荐给Java开发者们的一些值得一看的好书，从java基础开始到高级，以及从高级进阶到架构等的书单。

基本都是经典之作，可以利用工作闲暇的时间，系统阅读。

#### [java程序员进阶](http://youzhixueyuan.com/tag/java%e7%a8%8b%e5%ba%8f%e5%91%98%e8%bf%9b%e9%98%b6)之路



上图是完整[java程序员进阶](http://youzhixueyuan.com/tag/java%e7%a8%8b%e5%ba%8f%e5%91%98%e8%bf%9b%e9%98%b6)之路，可以进阶架构师也可以结合管理进阶，往技术经理、总监发展。

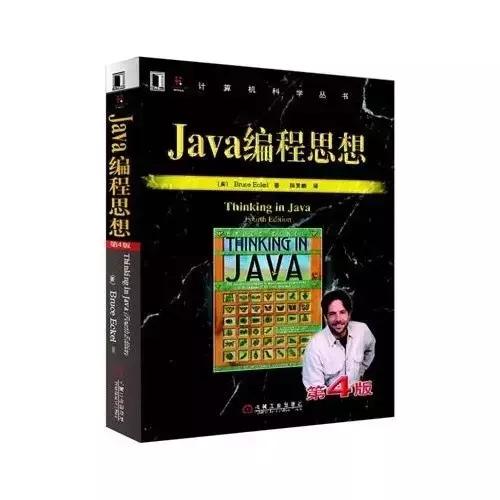
上周，写了程序员进阶到架构师三篇，基本涵盖了java程序员到架构师的必备技能。如果要想系统掌握，可以结合书单一起自学进阶。



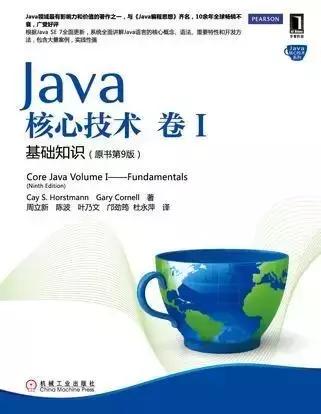
#### [java程序员进阶](http://youzhixueyuan.com/tag/java%e7%a8%8b%e5%ba%8f%e5%91%98%e8%bf%9b%e9%98%b6)书单

**1 刚毕业工作1-2年java程序员**

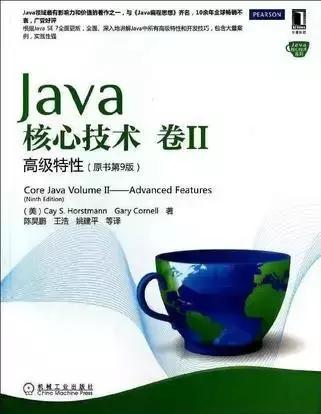
在这一年里，你至少需要看完《Java编程思想》这本书。如果部分章节，掌握不好的情况，就反复学习查看，直到掌握好为止。



《Thinking in Java》java系列经典开启书。Bruce Eckel在书中不仅仅讲解了Java的语法，更重要的是将OOP编程思想循序渐进、浅显易懂地表达出来。从这点上来说，中文版翻译成Java编程思想非常恰当



JAVA核心技术(卷1):基础知识,刚入行业的java程序员，很多都是以本书开启自己的生涯。与《Java编程思想》齐名的Java图书泰山北斗，前者内容丰富，在讲述Java语言的同时把Java编程思想也娓娓道来，让人如沐春风；后者偏重于讲解Java技术，举例说明了大量API的使用。



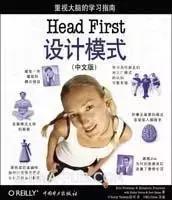
此书分为了基础版和高级特性版,你也可以读java编程思想，也有涉及高级特性部分。这本书在新手入门上比《Thinking in Java》稍胜一筹，但在内容深广度、性价比方面则跟《Thinking in Java》不可同日而语，建议选读。

这个阶段还需要掌握web开发基础：html css，js等web开发基础，以及java web开发框架 struts2 springmv mybatis等框架，熟练使用mysql以及相关工具mysql-front等，以及常用的开发工具：eclipse等。

**2 工作3年左右的java高级程序员**

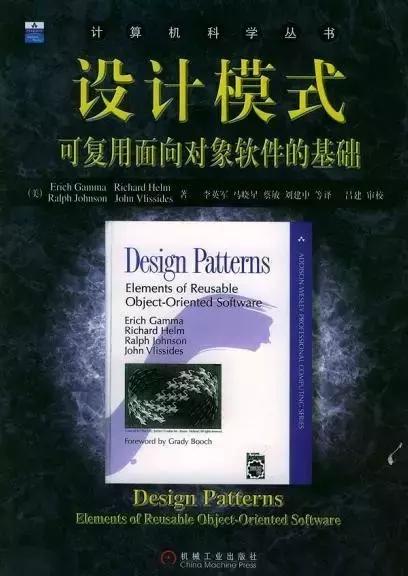
这部分时间段的同学，已经对Java有了一个更加深入的了解。但是对于面向对象的体会可能还不够深刻，编程的时候还停留在完成功能的层次，很少会去考虑设计的问题。于是这个时候，设计模式就来了。**这个阶段，你的核心任务就是提高你的代码能力，注重代码的质量。**

首先，从设计模式开始了解**代码这门艺术**。



《Head First设计模式》(中文版)共有14章，每章都介绍了几个设计模式，完整地涵盖了四人组版本全部23个设计模式。Strategy、Observer、Decorator、Abstract Factory、Factory Method、Singleton，Command、Adapter、Facade、TemplateMethod等,

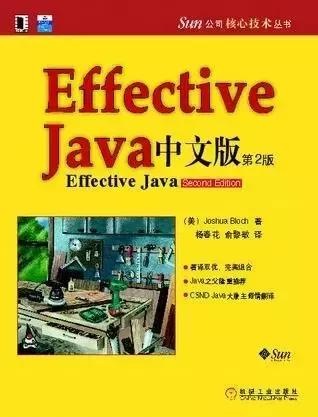
这本书，通俗易通，写得很有趣，图文并茂，设计模式入门。



号称设计模式领域的圣经，软件工程和软件架构必看的书籍。内功修养。这本书结合设计实作例从面向对象的设计中精选出23个设计模式，总结了面向对象设计中最有价值的经验，并且用简洁可复用的形式表达出来。

四位作者均是国际公认的面向对象软件领域的专家。Head First设计模式拥有风趣的语言和诙谐的例子，但它既缺乏实际的工程范例，也没有给出设计模式的应用/适用场景。

我个人建议是在读完Head First这本书之后立即阅读“四人帮”的这本设计模式。



本书介绍了在Java编程中78条极具实用价值的经验规则，这些经验规则涵盖了大多数开发人员每天所面临的问题的解决方案。通过对Java平台设计专家所使用的技术的全面描述，揭示了应该做什么，不应该做什么才能产生清晰、健壮和高效的代码。



可以适当的接触敏捷开发，在本书是全球的软件开发专家和软件工程大师Robert C.Martin将向您展示如何解决软件开发人员、项目经理及软件项目领导们所面临的最棘手的问题。这本综合性、实用性的敏捷开发和极限编程方面的指南，是由敏捷开发的创始人之一所撰写的。



Martin Fowler和《重构:改善既有代码的设计》另几位作者清楚揭示了重构过程，他们为面向对象软件开发所做的贡献，难以衡量。

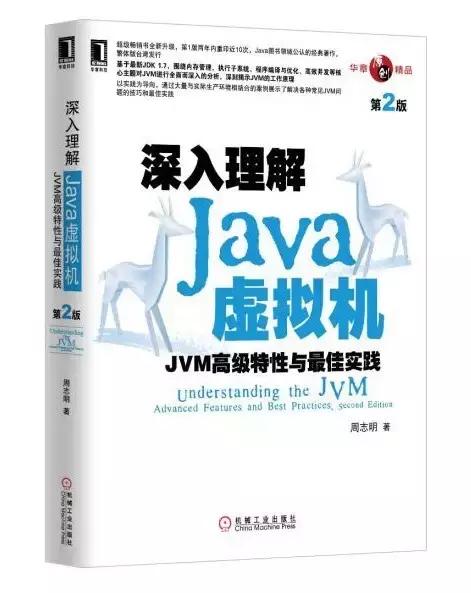
任何产品代码都不是一蹴而就，而是在反复不断的修改中进化而来。

重构正是这样一本介绍如何改进代码的书籍——如何在保持代码行为的基础上，提升代码的质量（这也是重构的定义）。

#### 3 工作3年以上准备往架构师方向发展

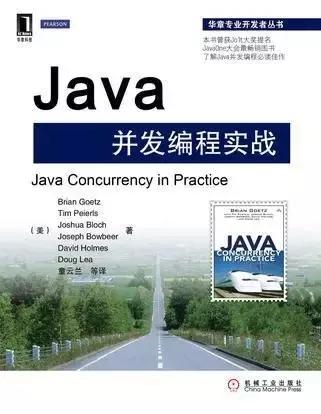
这个阶段，我用一句来总结。你需要做的是深入了解java相关的**技术原理**，Java底层和Java类库，以及SSM、SSH等常用框架源代码，JVM、分布式缓存、存储、Linux内核等的核心原理。

**Java程序员必读经典，深入理解Java虚拟机。**

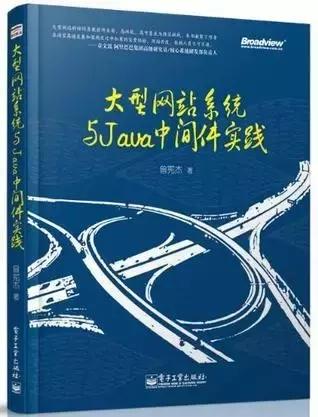


《深入理解Java虚拟机:JVM高级特性与最佳实践(第2版)》共分为五大部分，围绕内存管理、执行子系统、程序编译与优化、高效并发等核心主题对JVM进行了全面而深入的分析，深刻揭示了JVM的工作原理。

确为提升Java功力之良药。本书使自己对于虚拟机的运行机理与底层知识真正来了一次近距离接触，许多知识不再只是符号或文字的堆叠，而是一种真正地理解。国人很难写出这么棒的书，本书值得推荐。



其次，在过去的工作当中，你肯定或多或少接触过并发。这个时候，你应该去更加深入的了解并发相关的知识。推荐《Java并发编程实战》这本书。



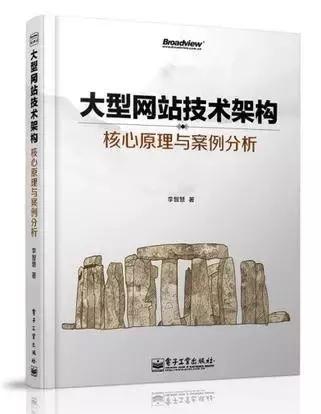
本书围绕大型网站和支撑大型网站架构的 Java 中间件的实践展开介绍。从分布式系统的知识切入，让读者对分布式系统有基本的了解；然后介绍大型网站随着数据量、访问量增长而发生的架构变迁；接着讲述构建 Java 中间件的相关知识；之后的几章都是根据笔者的经验来介绍支撑大型网站架构的 Java 中间件系统的设计和实践。

如果对淘宝的Java技术感兴趣，这本书就是最佳的干货选择。作者是淘宝Java中间件团队的创始人，如果你想全面了解中间件，推荐此书。



本书是作者在多年的工作中积累的第一手Spring框架使用经验的总结，深入剖析了Spring框架各个模块的功能、出现的背景、设计理念和设计原理，揭开了Spring框架的神秘面纱，使你“知其然，更知其所以然”。

Spring里实践了大量Java语言优秀的理念，此书通过讲解Spring原理，通俗易懂地向我们展示了这些理念。Spring对于java框架 系列的重要性之高，三大框架ssh里面写的最好的一本书，非常推荐这本书。



《大型网站技术架构：核心原理与案例分析》通过梳理大型网站技术发展历程，剖析大型网站技术架构模式，深入讲述大型互联网架构设计的核心原理，并通过一组典型网站技术架构设计案例，为读者呈现一幅包括技术选型、架构设计、性能优化、Web 安全、系统发布、运维监控等在内的大型网站开发全景视图。

大型网站架构设计的清单，比较全面、系统、清晰，虽然讲解得不是特别深入，但却比较朴实易懂，架构入门系列文章。



涵盖了Web站点性能优化的所有内容，包括数据的网络传输、服务器并发处理能力、动态网页缓存、动态网页静态化、应用层数据缓存、分布式缓存、Web服务器缓存、反向代理缓存、脚本解释速度、页面组件分离、浏览器本地缓存、浏览器并发请求、文件的分发、数据库I／O优化、数据库访问、数据库分布式设计、负载均衡、分布式文件系统、性能监控等。

高性能入门的好书，我个人也是从这本开启高性能之旅。

### 史上最全java架构师技能图谱（上）

architect-skills-map-1.html

#### 一：数据结构算法



**算法分析**

时间复杂度和空间复杂度

**算法思想**

递推、递归、穷举、贪心、分治、动态规划、迭代、分枝界限

**数据结构**

数组、链表、堆、栈、队列、Hash表、二叉树等

**算法**

排序

经典排序：插入排序、冒泡排序、快排（分划交换排序）、直接选择排序、堆排序、合并排序等

查找

经典查找：顺序查找、二分查找、二叉排序树查找

**高级算法**

贪婪

回溯

剪枝

动态规划

**大数据算法**

hash分桶

统计

#### 二：Java进阶



**java编程基础：**

对象和类 、基本数据类型 、变量类型、运算符、循环分支结构、数组、正则表达式等

**集合**



总的说来，Java API中所用的集合类，都是实现了Collection接口，他的一个类继承结构如下：

Collection<–List<–Vector

Collection<–List<–ArrayList

Collection<–List<–LinkedList

Collection<–Set<–HashSet

Collection<–Set<–HashSet<–LinkedHashSet

Collection<–Set<–SortedSet<–TreeSet



**面向对象高级知识**



类、对象、继承、构造函数、封装、接口、抽象类、多态、重写、this static关键字、类与对象的关系

**异常处理**

异常类类图：throwable exception error RuntimeException

异常处理机制

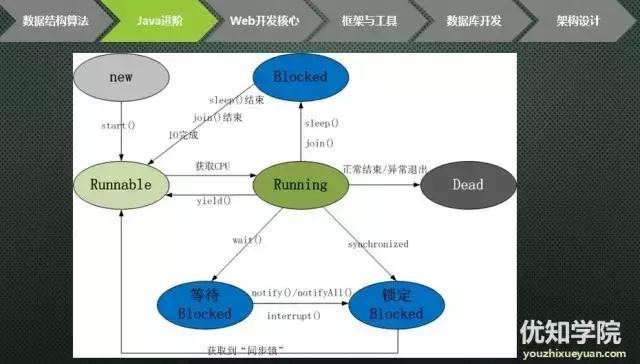
如何定义和使用异常

运行时异常和受检查异常区别

运行时错误

java异常处理的原则和技巧

**多线程**



概念与原理

创建于启动

线程的生命周期及五种基本状态

线程交互

死锁

调度合并

调度让步

调度休眠

同步方法

同步块

同步与锁

线程池

阻塞队列

**IO/NIO**

同步阻塞 同步非阻塞 异步IO

**反射**

**序列化**

**泛型**

**网络编程**

**高级特性**

**JVM**

**运行时数据区**：方法区、虚拟机栈、本地方法栈、堆、程序计算器

**GC算法：**

内参回收三要素：什么内容需要回收、什么时候回收、如何回收

并发与执行

引用计数算法

根搜索算法

垃圾回收算法：标记-清楚算法 复制算法 标记-整理算法 分代手机算法

垃圾收集器：新生代、老年代收集器

**溢出**

java堆溢出

方法区溢出

outofmemoryerror

虚拟机栈和本地方法栈溢出

直接内容溢出

#### 三：Web开发核心



**HTML JS CSS**

html js css语法基础

Js css框架

Html开发工具

JS和CSS调试工具

**模板引擎**

jsp

velocity

freemarker

**Java web**

容器：tomcat jetty等

热部署插件：run-jetty-run

cookie session使用和区别

fliter和listener的启动和步骤

身份验证

单点登录原理以及实现

**web核心**

事物JTA

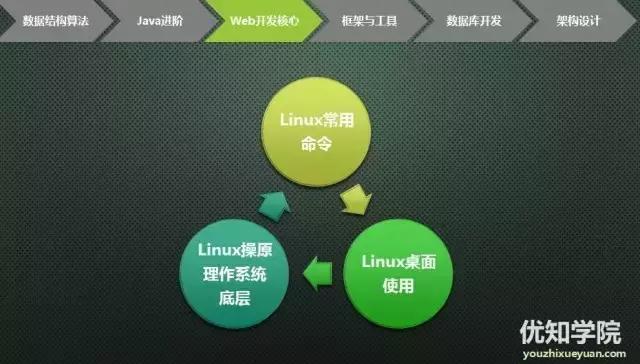
JMX

安全：JCCA/JAAS

通信:JNDI/JMS

SSI技术

**linux**



常用命令以及操作系统原理等

**线上故障处理和分析**



**性能工具**

visualVM Jprofiler JMeter等

**线上故障**

线程数超标

访问超时

长事务

CPU超标

内存超标

**开发工具使用**



**web开发调试**

firebug

Web Developer

JavaScript Debugger

IETester

Yslow

**构建工具**

maven Grails

maven私服 nexus

**版本控制**

git svn

**java调试工具**

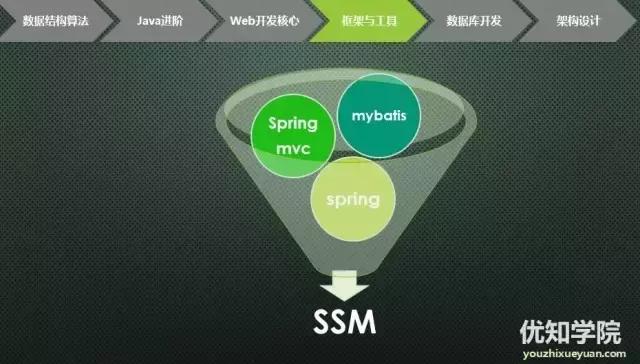
JCover

Junit

Jtest

以及大量的eclipse插件，eg：findbugs等

#### 开发框架



SSH：struts2+spring+hibernate

SSM:springmvc+spring+mybatis

**阿里开源框架**



#### [架构师进阶](http://youzhixueyuan.com/tag/%e6%9e%b6%e6%9e%84%e5%b8%88%e8%bf%9b%e9%98%b6)六大要求



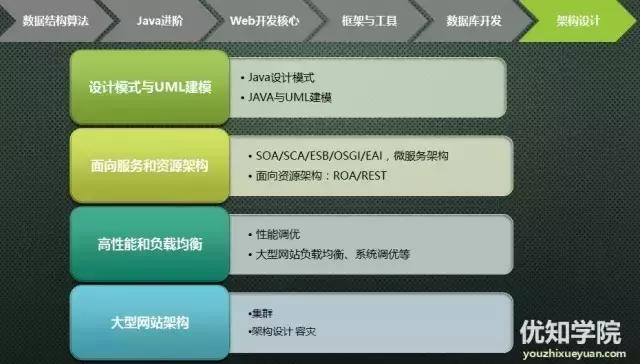
* 程序设计要求
* WEB开发要求
* 架构设计要求
* 运维能力要求
* 操作系统要求
* 产品业务要求

#### 数据库技能



* 理论基础
* 数据库设计原则和范式
* 事务(ACID,事务的隔离，锁，传播机制)
* 数据库优缺点
* Mysql oracle sqlserver
* Nosql (redis，mongoDB，memcached,hbase)
* Sql语句
* 数据库创建，权限分配，表的创建，增删改查，连接，子查询
* 触发器、存储过程、事务控制
* 性能优化&分库分表等
* 索引原理及适用，大表查询优化，多表连接查询优化，子查询优化等
* 分库、分表，冷备热备，主从备份、双机热备、纵向扩展、横向扩展

#### 架构设计



* 设计模式与UML建模
* Java设计模式
* JAVA与UML建模
* 面向服务和资源架构
* SOA/SCA/ESB/OSGI/EAI，微服务架构
* 面向资源架构：ROA/REST
* 高性能和负载均衡
* 性能调优
* 大型网站负载均衡、系统调优等
* 大型网站架构
* 集群
* 架构设计 容灾

#### 设计模式



* 结构型模式
* 创建型模式
* 行为模式等
* 重用性高
* 易维护

#### 技术架构能力



* 缓存搭建
* CDN基站搭建
* 消息队列搭建
* 数据库拆分
* 工程代码拆分
* SOA服务
* 分布式文件
* 搜索引擎
* Sql和Nosql混搭
* 运维系统
* 监控系统
* 机房容灾

#### 架构前瞻性



•业务和技术的把控能力

•技术挑战

•预判业务

#### 淘宝店铺装修项目大数据实战



如果你对java架构有兴趣可以参加优知学院的java进阶实战，**java进阶群：629511325**

淘宝店铺装修大数据实战

业务场景：涉及淘宝大数据业务的访问，以及大量的读写业务场景。

技术场景：涉及分布式场景(硬软件集群和负载均衡等)、dubbo(SOA)、消息队列(ActiveMQ)、分布式缓存Redis、分布式存储(TFS)等技术场景。

* 架构设计
* 详细技术方案
* 数据库设计
* 表结构搭建
* 搭建SSM框架
* 数据库以及开发环境搭建
* 搭建分布式缓存和分布式文件存储系统
* 编码阶段

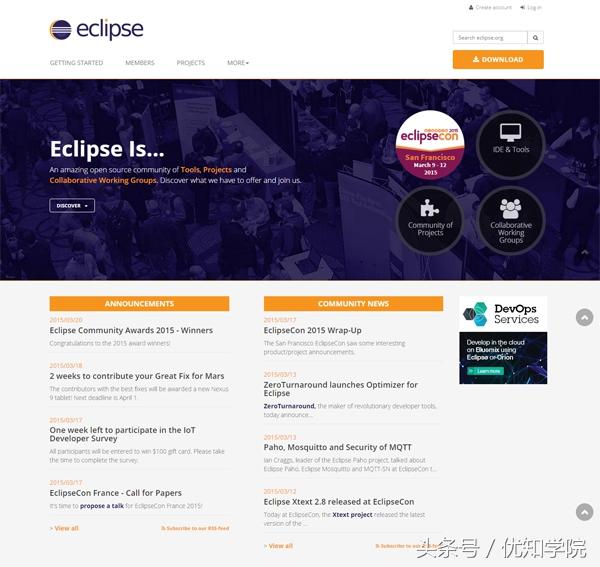
## java程序员的9款开发利器：开发环境+性能调优+代码调试+持续集成工具

development-tools-for-java-programmers.html

今天推荐java程序员开发利器，包含如如下：

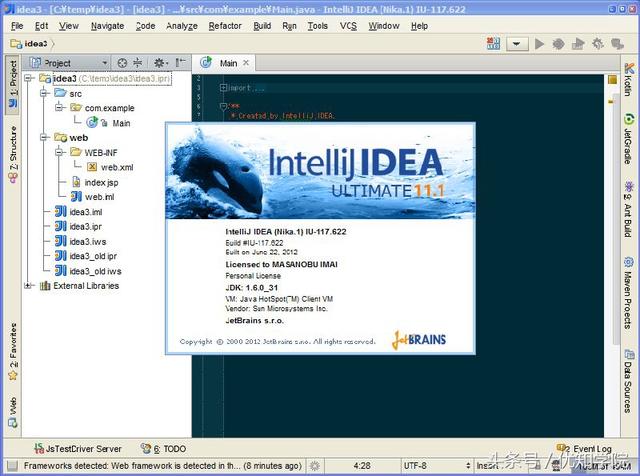
### 开发环境：

**Eclipse**



**IntelliJ IDEA**

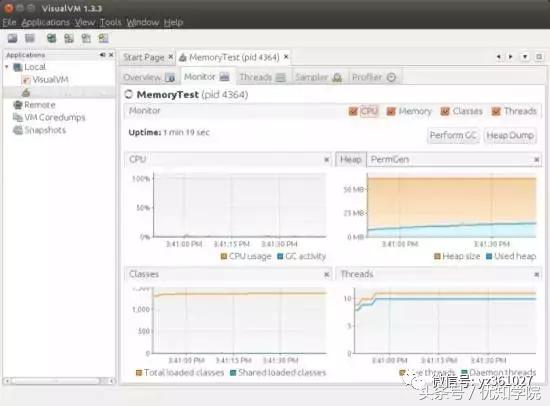
IntelliJ在业界被公认为最好的java开发工具之一，尤其在智能代码助手、代码自动提示、重构、J2EE支持、各类版本工具(git、svn等)、JUnit、CVS整合、代码分析、 创新的GUI设计等方面的功能可以说是超常的。



### 性能调优

**JVM调优：VisualVM**

VisualVM 是一款免费的，集成了多个 JDK 命令行工具的可视化工具，它能为您提供强大的分析能力，对 Java 应用程序做性能分析和调优。这些功能包括生成和分析海量数据、跟踪内存泄漏、监控垃圾回收器、执行内存和 CPU 分析，同时它还支持在 MBeans 上进行浏览和操作。本文主要介绍如何使用 VisualVM 进行性能分析及调优。



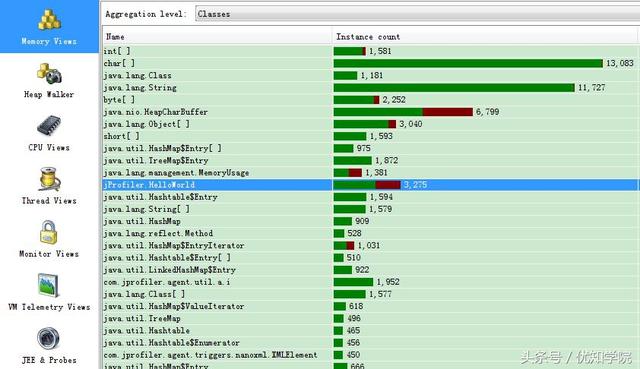
**JProfiler**

JProfiler是由ej-technologies GmbH公司开发的一款性能瓶颈分析工具(该公司还开发部署工具)。

其特点:

* 使用方便
* 界面操作友好
* 对被分析的应用影响小
* CPU,Thread,Memory分析功能尤其强大
* 支持对jdbc,noSql, jsp, servlet, socket等进行分析
* 支持多种模式(离线，在线)的分析
* 跨平台





**JMeter**

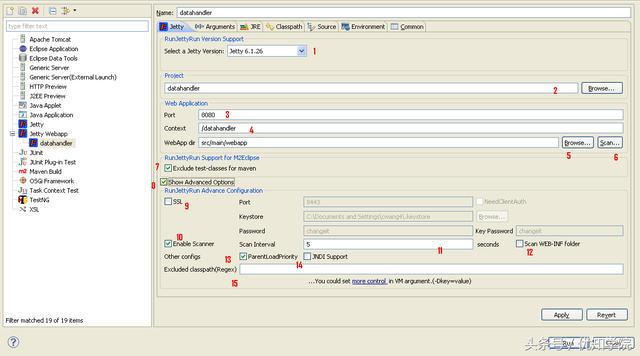
Apache jmeter 是一个100％的纯java桌面应用，用于压力测试和性能测量。它最初被设计用于Web应用测试但后来扩展到其他测试领域。

Apache jmeter 可以用于对静态的和动态的资源（文件，Servlet，Perl脚本，java 对象，数据库和查询，FTP服务器等等）的性能进行测试。它可以用于对服务器，网络或对象模拟繁重的负载来测试它们的强度或分析不同压力类型下的整体性能。你可以使用它做性能的图形分析或在大并发负载测试你的服务器/脚本/对象。



### 代码调试

Jetty 与 Eclipse 的集成，runjettyrun：热部署必备



Java Decompiler 可以反编译（查看）里面的源代码，调试必备

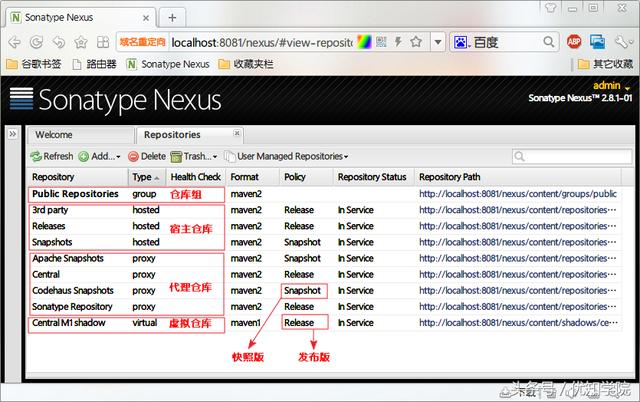


### 持续集成:jenkins

Jenkins是一个功能强大的应用程序，允许持续集成和持续交付项目，无论用的是什么平台。这是一个免费的源代码，可以处理任何类型的构建或持续集成。集成Jenkins可以用于一些测试和部署技术。



### Sonatype Nexus 搭建maven私服



当然，还有很多开发工具，也欢迎交流里说出你最喜欢的工具。